SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(7), 727–765 (JULY 1995)

Experimental Results on String Matching Algorithms

thierry lecroq
Laboratoire d'Informatique de Rouen, Université de Rouen, Facultés des Sciences, Place Emile Blondel, 76821 Mont-Saint-Aignan Cedex, France (email: lecroq@dir.univ-rouen.fr)

SUMMARY

We present experimental results for string matching algorithms which are known to be fast in practice. We compare these algorithms through two aspects: the number of text character inspections and the running time. These experiments show that for large alphabets and small patterns the Quick Search algorithm of Sunday is the most efficient and that for small alphabets and large patterns it is the Reverse Factor algorithm of Crochemore *et al.* which is the most efficient.

key words: string searching; pattern matching; text editing; algorithms on words

INTRODUCTION

The string-matching problem consists of finding all occurrences of a pattern x of length m in a text y of length n. The pattern and the text are both words built on the same alphabet Σ. The brute-force algorithm to solve this problem has a quadratic $O(nm)$ worst-case time complexity. Fortunately there exist several linear-time algorithms that have been developed in the last twenty years. Some of them are even sublinear in practice, but Rivest[1] proved in 1977 that at least $n - m + 1$ text characters must be inspected in the worst case.

The best way to understand how a string matching algorithm works is to imagine that there is a window on the text. This window has the same length as the pattern x. This window is first aligned with the left end of the text and then the string matching algorithm scans if the characters of the window match the characters of the pattern (we will call that specific work an *attempt*). After each attempt the window is shifted to the right over the text until it goes over the right end of the text. A string matching algorithm is a succession of attempts and shifts. The aim of a good algorithm is to minimize the work done during each attempt and to maximize the length of the shifts.

Most of the linear string matching algorithms preprocess the pattern before the search phase. Only one algorithm, designed by Crochemore,[2] has a linear worst-case time complexity without a preprocessing phase.

The word done during the preprocessing phase usually helps the algorithm to maximize the length of the shifts. The preprocessing phase generally needs an extra

CCC 0038–0644/95/070727–39

Received April 1994
Revised September 1994

space which is linear in the length of the pattern. Only two algorithms[3,4] need a constant extra space. They both have the particularity that they scan the characters in the window in the two directions: one part from left to right and the other part from right to left.

The general scheme of a string matching algorithm is given in Figure 1.

There are different ways to check if the characters of the window match the characters of the pattern. After a mismatch or a complete match of the whole pattern the brute-force algorithm shifts the window to the right by exactly one position. The direct improvements of the brute-force algorithm scan the characters of the window from left to right: they are Morris and Pratt's algorithm,[5] Knuth, Morris and Pratt's algorithm[6] and Simon's algorithm.[7] These algorithms scan in the worst case at most $2n$ text characters. They use a shift function based on the notion of periods of the prefixes of the pattern.

The Quick Search algorithm of Sunday[8] also scans the characters of the window from left to right, but it uses a 'heuristic' shift function called the occurrence shift, first introduced by Boyer and Moore.[9]

The algorithms which scan the characters of the window from right to left are well known for their fast running time. The first of them was the famous Boyer–Moore algorithm[9] which was improved by Galil,[10] Apostolico and Giancarlo[11] and Crochemore *et al.*[12] These algorithms use two shift functions: the occurrence shift and the matching shift. There exists a simplification of the Boyer–Moore algorithm, due to Horspool,[13] which uses only the occurrence shift.

There are also two algorithms which scan the characters of the window from right

```
ALGORITHM STRING-MATCHING;
/* find occurrences of x[1,m] in y[1,n] */
Begin
    - preprocessing phase;
    - search phase:
        align the left end of the window with the left end of y;
        while the right end of the window has not gone over the right end of y
        do
            attempt;
            if an occurrence of x has been found then
                report it;
            endif
            shift;
        endwhile
End.
```

Figure 1. General scheme of a string-matching algorithm

```
ALGORITHM BF;
Begin
    i:=0;
    while i ≤ n-m do
        j:=1;
        while j ≤ m andif x[j] = y[i+j] do
            j:=j+1;
        endwhile
        if j > m then
            output(i+1);
        endif
        i:=i+1;          /* shift */
    endwhile
End.
```

Figure 2. The brute-force algorithm

to left using the smallest suffix automaton (or dawg) of the reverse pattern, they are called the Reverse Factor algorithm and the Turbo Reverse Factor algorithm.[12,14]

Some algorithms scan the characters of the window in the two directions. We can distinguish two types of such algorithms. First those which scan from one direction and then from the other from a specific position into the pattern: this is the case for the already cited algorithms of Galil and Seiferas[3] and Crochemore and Perrin.[4] The position in the pattern is chosen from some combinatorial properties on words. The second type of algorithm partitions the positions in the pattern into two subsets and first scans the position of a subset from left to right and then the positions of the other subset from right to left. They are the algorithm of Colussi[15] and the algorithm of Galil and Giancarlo[16] which in the worst case scan at most $3n/2$ and $4n/3$ text characters, respectively.

Several experiments on string matching algorithms have already been reported.[13,17–21] In this paper we report experiments on eight different algorithms: the brute force algorithm, the Boyer–Moore algorithm, the Apostolico–Giancarlo algorithm, the Turbo-BM algorithm, the Boyer–Moore–Horspool algorithm, the Quick Search algorithm, the Reverse Factor algorithm and the Turbo Reverse Factor algorithm (respectively BF, BM, AG, TBM, BMH, QS, RF and TRF for short).

This paper is organized as follows: in the second section we present the algorithms tested in the following sections. In the third section we present the different texts used. In the fourth section we give results for the number of inspections of text characters, and in the fifth section we exhibit results for the running times.

We denote by per(w) the smallest period of a word w.

We identify each attempt with the integer $0 \leq i \leq n - m$ such that the window is composed of the characters $y[i + 1,i + m]$.

In all the algorithms below i will be an index in the text y and j will be an index in the pattern x.

Brute force

The brute-force algorithm may scan the characters of the window from left to right beginning with the leftmost character of the window or from right to left beginning with the rightmost character of the window. In any case, after a mismatch or a complete match of the entire pattern it shifts the window of exactly one position to the right. We give the version which scans the characters from left to right in Figure 2.

The BF algorithm has a quadratic $O(nm)$ worst-case time complexity. The average number of comparison for one text character is $1 + 1/(|\Sigma| - 1)$.

Boyer–Moore

The Boyer–Moore algorithm[9] scans the characters of the window from right to left beginning with the rightmost character of the window. In case of a mismatch or a complete match of the whole pattern it uses two shift functions to shift the window to the right. These two functions are called the *occurrence shift* and the *matching shift*.

If a mismatch occurs between a character $x[j]$ and a character $y[i + j]$ during an attempt i of the BM algorithm then the occurrence shift consists in aligning $y[i + j]$ with its rightmost occurrence in $x[1,m - 1]$ (see Figure 3). If $y[i + j]$ does not appear in x then no occurrence of x in y can include $y[i + j]$ and the left end of the pattern can be shifted immediately to the right of $y[i + j]$. The matching shift consists of aligning the factor $y[i + j + 1,i + m]$ with its rightmost occurrence in $x[1,m]$ preceded by a character different from $x[j]$ (see Figure 4) or if it is not possible to align the longest suffix of $y[i + j + 1,i + m]$ with a matching prefix of x.

The BM algorithm then applies the maximum between the occurrence shift and the matching shift (see Figure 5).

These two shift functions are defined as follows:

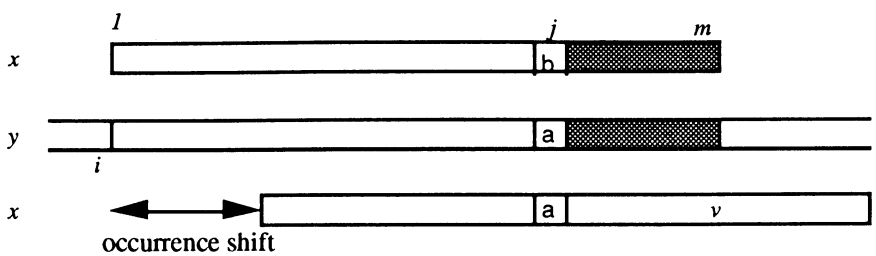

Figure 3. The occurrence shift; the character a *does not appear in the suffix* v *of* x

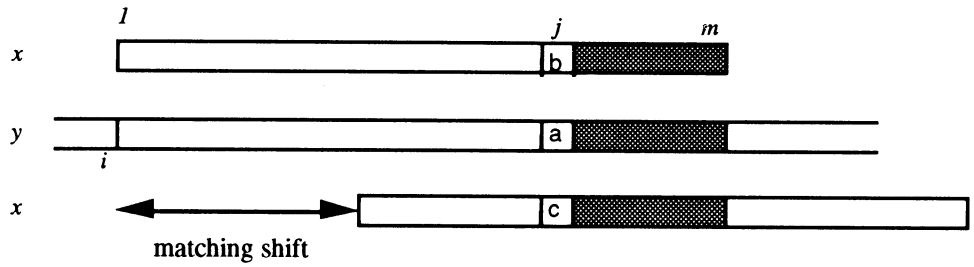

Figure 4. The matching shift, a≠b *and* c≠b

```
ALGORITHM BM;
Begin
    i:=0;
    while i ≤ n-m do
        j:=m;
        while j > 0 andif x[j] = y[i+j] do
            j:=j-1;
        endwhile
        if j ≤ 0 then
            output(i+1);
            i:=i+matching_shift(0);          /* shift of the length of the period of x*/
        else
            i:=i+max(occurrence_shift(y[i+j])-m+j,matching_shift(j));    /* shift */
        endif
    endwhile
End.
```

Figure 5. The Boyer–Moore algorithm

$\forall$a $\in \Sigma$, occurrence_shift(a) = min $\{j/1 \leq j < m$ and $x[m-j]$ = a$\}$ if a appears in x,
occurrence_shift(a) = m otherwise
for $1 \leq j \leq m$, matching_shift (j) = min $\{h/(h \geq j$ and $x[1,m-h] = x[h+1,m])$ or
$(h < j$ and $x[j-h] \neq x[j]$ and $x[j-h+1,m-h] = x[j+1,m])\}$

and we define matching_shift(0) = per(x).

The occurrence shift and the matching shift functions are preprocessed in time $O(m + |\Sigma|)$ before the search phase and require an extra space in $O(m + |\Sigma|)$.

The first correct published version of the computation of the matching shift was due to Rytter[22] and another version is due to Mehlhorn.[23]

When searching for all the occurrences of the pattern in the text the BM algorithm has a quadratic worst-case time complexity. The exact complexity[6] is $O(n + rm)$ where r is the number of occurrences of x in y. Galil[10] gave in 1979 a version of the Boyer–Moore algorithm which runs linearly independently of r.

The proof of the linearity of the BM algorithm when searching for the first occurrence of the pattern is not trivial. A first proof of $7n$ characters comparisons was given in 1977 by Knuth.[6] In 1980 Guibas and Odlyzko[24] gave a proof of $4n$ comparisons and conjectured that the real bound was $2n$ comparisons. It was only in 1990 that Cole[25] gave the tight bound of $3n - n/m$ comparisons for non-periodic patterns.

Apostolico–Giancarlo

The Boyer–Moore algorithm is difficult to analyse because after each attempt it forgets all the characters it has already matched. In 1986 Apostolico and Giancarlo[11] designed an algorithm which remembers each factor of the text which is a suffix of the pattern.

Let us assume that during an attempt $< i$ the algorithm has matched a suffix of x of length k at position $i + j$ with $0 < j < m$, then $\mathsf{skip}[i + j]$ is equal to k. During the attempt i if the algorithm compares successfully the factor of the text $y[i + j + 1, i + m]$ (see Figure 6), then

1. If $x[j - k + 1, j]$ is a suffix of x it is possible to jump over the factor of the text $y[i + j - k + 1, i + j]$ and to resume the comparisons with the characters $y[i + j - k]$ and $x[j - k]$.
2. If $x[j - k + 1, j]$ is not a suffix of x we know that no occurrence of the pattern can be found during attempt i and the algorithm must perform a shift.

Another case can occur if $k > j$ (see Figure 7), then it is sufficient to know if $x[1, j]$ is a suffix of x.

In each case the only information which is needed is the length of the longest suffix of x ending at position j in x. For that the AG algorithm use two data structures:

(a) a table skip which is updated at the end of each attempt i in the following way:

$$skip[i + m] = \max \{k / x[m - k + 1, m] = y[i + m - k + 1, i + m]\}$$

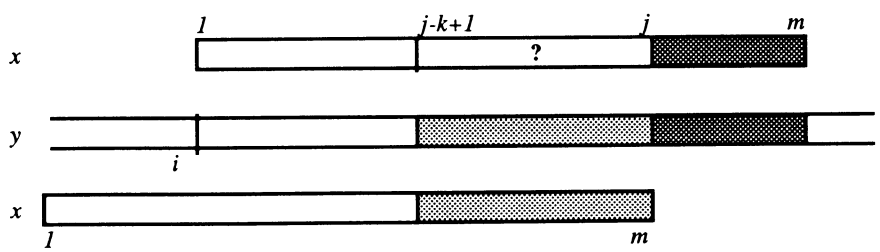

Figure 6. Typical situation during the Apostolico–Giancarlo algorithm (I). The factors coloured with the same gray match

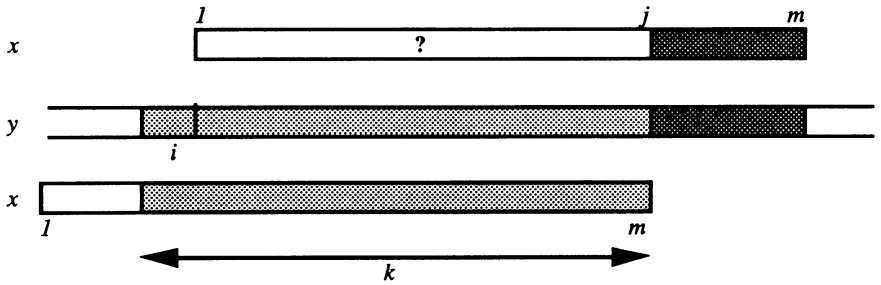

Figure 7. Typical situation during the Apostolico–Giancarlo algorithm (II)

(b) a table suffix whose values are preprocessed in the following way:

$$\text{for } 1 \leq j < m,\ suffix[j] = \max\{k / x[j-k+1, j] = x[m-k+1, m]\}$$

During an attempt i, at position $i+j$ if $\text{skip}[i+j] \neq 0$ and $\text{suffix}[j] \geq \text{skip}[i+j]$ or $\text{suffix}[j] = j$ then it is possible to jump over the factor of the text $y[i+j-\text{skip}[i+j]+1, i+j]$.

If $\text{skip}[i+j] \neq 0$ and $\text{suffix}[j] < \text{skip}[i+j]$ and $j > \text{suffix}[j]$ then we know that a mismatch would occur between characters $y[i+j-\text{suffix}[j]]$ and $x[j-\text{suffix}[j]]$ and we can compute the shift with max (occurrence_shift($y[i+j-\text{suffix}[j]] - m + j + \text{suffix}[j]$, matching_shift($j - \text{suffix}[j]$] (see Figure 8). This is a slight modification of the algorithm presented in Reference 1, where in that case the shift is computed only with matching_shift′(j), where

$$\text{for } 1 \leq j \leq m,\ \text{matching\_shift}(j) = \min\{h / (h \geq j \text{ and } x[1, m-h] = x[h+, m]) \text{ or } (h < j \text{ and } x[j-h+1, m-h] = x[j+1, m])\}$$

This modification gives a longer shift and makes the algorithm a little bit faster in practice but does not modify the worst-case time complexity of the algorithm.

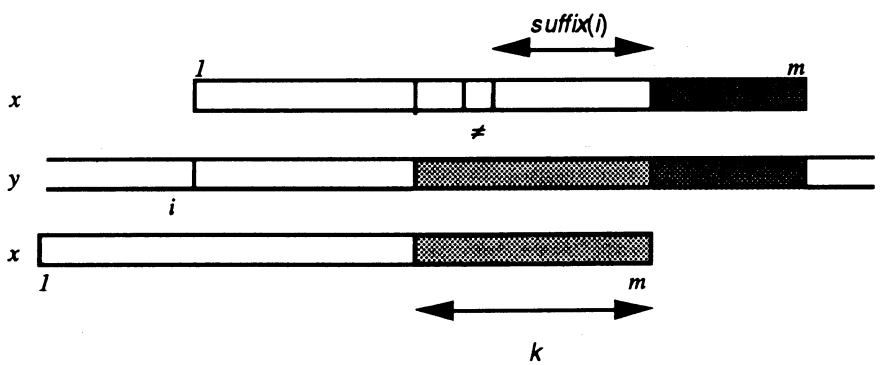

Figure 8. Mismatch in the Apostolico–Giancarlo algorithm

The table suffix can be precomputed in time $O(m)$. Then the complexity in space and time of the AG algorithm is the same as that for the Boyer–Moore algorithm: $O(m + |\Sigma|)$.

The AG algorithm is presented in Figure 9.

During the search phase only the last m informations of the table skip are needed at each attempt, so the size of the table skip can be reduced to $O(m)$.

The Apostolico–Giancarlo algorithm performs in the worst case at most $2n - m + 1$ comparisons of text characters.

```
ALGORITHM AG;
Begin
    for k:=1 to n do
        skip[k]:=0;
    endfor
    i:=0;
    while i ≤ n-m do
        j:=m;
        while j > 0 and ((skip[i+j] ≠ 0 and (suffix[j] ≥ skip[i+j] or
              j ≤ suffix[j])) or (skip[i+j] = 0 and x[j] = y[i+j])) do
            j:=j-max(1,skip[i+j]);
        endwhile
        if j ≤ 0 then
            output(i+1);
            skip[i+m]:=m;
            i:=i+matching_shift(0);
        else
            if skip[i+j] ≠ 0 then
                j:=j-suffix[j];
            endif
            skip[i+m]:=m-j;                    /* memorization */
            i:=i+max(occurrence_shift(y[i+j])-m+j,matching_shift(j));    /* shift */
        endif
    endwhile
End.
```

Figure 9. The Apostolico–Giancarlo algorithm

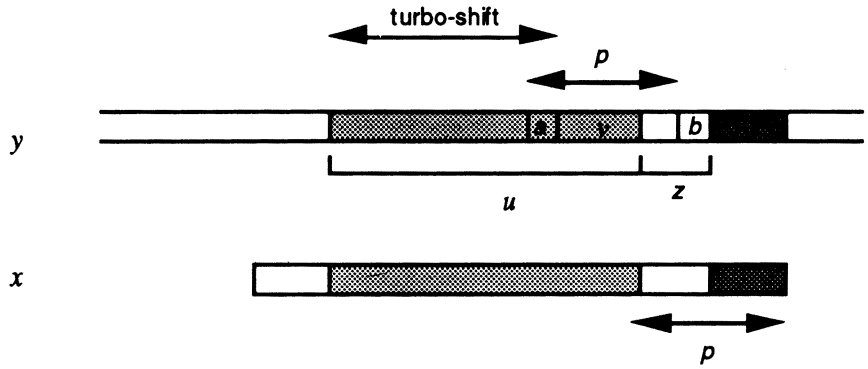

Figure 10. A turbo-shift can apply when $|v| < |u|$

Turbo-BM

The Turbo-BM algorithm[12] can be viewed as a simplification of the Apostolico–Giancarlo algorithm. It needs no extra preprocessing and requires only a constant extra space compared to the original Boyer–Moore algorithm. It consists of remembering only the factor of the text that matched a suffix of the pattern during the last attempt (and only if a matching shift has been performed). This technique has two advantages: (i) it is possible to jump over this factor and (ii) it can enable a turbo-shift to be performed.

A turbo-shift can occur if during the current attempt the suffix of the pattern that matches the text is shorter than the one remembered from the preceding attempt. In this case (see Figure 10) let us call u the remembered factor and v the suffix matched during the current attempt, such that uzv is a suffix of x. Then av is a suffix of x, the two characters a and b occur at distance p in the text, and the suffix of x of length $|uzv|$ has a period of length p, and thus it cannot overlap both occurrences of characters a and b in the text. The smallest shift possible has length $|u| - |v|$, which we call a turbo-shift.

Always in the case where $|v| < |u|$, if the occurrence shift is the larger shift then the actual shift must be greater or equal to $|u| + 1$ (see Figure 11). In this case the two characters c and d are different because we assumed that the previous shift was a matching shift. Then a shift greater than the turbo-shift but smaller than $|u| + 1$ would align c and d with the same character in v. Thus if the occurrence shift is the larger shift the length of the actual shift must be at least equal to $|u| + 1$.

The TBM algorithm is depicted in Figure 12. The variable u memorizes the length of the suffix matched during the previous attempt and the variable v memorizes the length of the suffix matched during the current attempt.

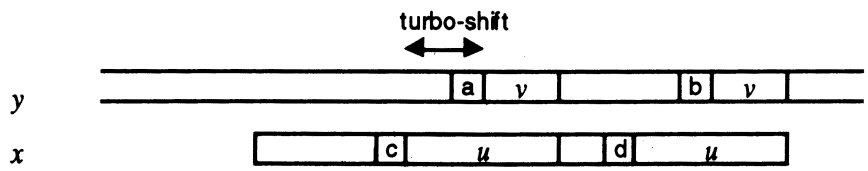

Figure 11. c ≠ d *so they cannot be aligned with the same character in* v

```
ALGORITHM TBM;
Begin
    i:=0;
    u:=0;
    shift:=m;
    while i ≤ n-m do
        j:=m;
        while j > 0 andif x[j] = y[i+j] do
            if u ≠ 0 and j = m-shift then
                j:=j-u;
            else
                j:=j-1;
            endif
        endwhile
        if j ≤ 0 then
            output(i+1);
            shift:=matching_shift(0);
            u:=m-shift;
        else
            v:=m-j;
            turbo_shift:=u-v;
            shift:=max(occurrence_shift(y[i+j])-m+j,matching_shift(j),turbo_shift);
            if shift = matching_shift(j) then
                u:=min(m-shift,v);      /* memorization */
            else
                if turbo_shift < occurrence_shift(y[i+j])-m+j then
                    shift:=max(shift,u+1);
                endif
                u:=0;
            endif
        endif
        i:=i+shift;                     /* shift */
    endwhile
End.
```

Figure 12. The Turbo-BM algorithm

The number of character comparisons performed by the TBM algorithm is bounded by $2n$.

Boyer–Moore–Horspool

In 1980 Horspool proposed to use only the occurrence shift of the rightmost character of the window to compute the shifts in the Boyer–Moore algorithm.

The Boyer–Moore–Horspool algorithm is shown in Figure 13.

The average number of comparisons performed by the BMH algorithm for one text character is between $1/|\Sigma|$ and $2/(|\Sigma|+1)$.[18]

Quick Search

In 1990 Sunday[8] designed an algorithm which scans the characters of the window from left to right beginning with the leftmost character of the window, and computes its shifts with the occurrence shift of the character of the text immediately after the right end of the window.

We now define the occurrence shift as follows:

$$\forall\ a \in \Sigma,\ \text{QS\_occurrence\_shift(a)} = \min\ \{j/1 \leq j \leq m \text{ and } x[m-j] = a\}$$
if a appears in x,
$\text{QS\_occurrence\_shift(a)} = m$ otherwise

This modification enables the last character of the pattern to be recognised.

```
ALGORITHM BMH;
Begin
    i:=0;
    while i ≤ n-m do
        j:=m;
        while j > 0 andif x[j] = y[i+j] do
            j:=j-1;
        endwhile
        if j ≤ 0 then
            output(i+1);
        endif
        i:=i+occurrence_shift(y[i+m]);          /* shift */
    endwhile
End.
```

Figure 13. The Boyer–Moore–Horspool algorithm

```
ALGORITHM QS;
Begin
    i:=0;
    while i ≤ n-m do
        j:=1;
        while j ≤ m andif x[j] = y[i+j] do
            j:=j+1;
        endwhile
        if j > m then
            output(i+1);
        endif
        i:=i+QS_occurrence_shift(y[i+m+1])+1;          /* shift */
    endwhile
End.
```

Figure 14. The Quick Search algorithm

Reverse Factor

The Boyer–Moore type algorithms match some suffixes of the pattern, but it is possible to match some prefixes of the pattern reading the character of the window from right to left and then improve the length of the shifts. This is made possible by the use of the smallest suffix automaton (also called dawg for directed acyclic word graph) of the reverse pattern. This algorithm is called the Reverse Factor algorithm.

The smallest suffix automaton of a word w[27,28] is a deterministic finite automaton $\mathcal{A} = (S, s_0, F, \delta)$ where S is a set of states, $s_0 \in S$ is the start state, F is the set of the final states and $\delta : S \times \Sigma \rightarrow S$ is the transition function. The language accepted by $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \{u \in \Sigma^* / \exists\ v \in \Sigma^*$ such that $w = vu\}$.

The preprocessing phase of the RF algorithm consists of computing the smallest suffix automaton for the reverse pattern x^R. It is linear in time and space in the length of the pattern. During the search phase the RF algorithm parses the characters of the window from right to left with the automaton $\mathcal{A}$ starting with state s_0. It goes until there are no more transitions defined for the current character of the window from the current state of the automaton or if the whole window has been scanned. At this moment it is easy to know what is the length of the longest prefix of the pattern which has been matched: it corresponds to the length of the path taken in $\mathcal{A}$ from the start state s_0 and the last final state encountered. Knowing the length of this longest prefix, it is trivial to compute the right shift to perform.

The RF algorithm is shown in Figure 15.

The RF algorithm has a quadratic worst-case time complexity but it is optimal on the average. It performs $O(n\log_{|\Sigma|}m)/m)$ inspections of text characters on the average, reaching the best bound shown by Yao[29] in 1979.

```
ALGORITHM RF;
Begin
    i:=0;
    while i ≤ n-m do
        j:=m;
        state:=s0;
        shift:=m;
        while δ(state,y[i+j]) is defined do
            state:=δ(state,y[i+j]);
            if state ∈ F then
                shift:=j;
            endif
            j:=j-1;
        endwhile
        if j ≤ 0 then
            output(i+1);
            shift:=per(x);
        endif
        i:=i+shift;     /* shift */
    endwhile
End.
```

Figure 15. The Reverse Factor algorithm

Turbo Reverse Factor

It is possible to make the Reverse Factor algorithm linear. It is in fact sufficient to remember the prefix u matched during the last attempt. Then during the current attempt when reaching the right end of u, it is easy to show that it is sufficient to read again at most the rightmost half of u.[10,23] This is done by the Turbo Reverse Factor algorithm.

If a word z is a factor of a word w we define disp(z,w), the displacement of z in w, to be the least integer $d > 0$ such that $w[m-d-|z|-1, m-d] = z$.

The general situation of the TRF algorithm is when a prefix u has been found in the text during the last attempt and for the current attempt the algorithm tries to match the factor v of length $m-|u|$ in the text immediately at the right of u. The point between u and v is called the decision point. If v is not a factor of x then the shift is computed as in the Reverse Factor algorithm. If v is a suffix of x then

an occurrence of x has been found. If v is not a suffix but a factor of x then it is sufficient to scan again the min $(\mathrm{per}(u), |u|/2)$ rightmost characters of u.

If u is periodic (i.e. $\mathrm{per}(u) \leq |u|/2$) let z be the suffix of u of length $\mathrm{per}(u)$. By definition of the period z is an acyclic word, and then an overlap such as shown in Figure 16 is impossible. Thus, z can only occur in u at distances which are multiples of $\mathrm{per}(u)$, which implies that the smallest proper suffix of uv which is a prefix of x has a length equal to $|uv| - \mathrm{disp}(zv,x) = m - \mathrm{disp}(zv,x)$. Thus the length of the shift to perform is $\mathrm{disp}(zv,x)$.

If u is not periodic ($\mathrm{per}(u) > |u|/2$), it is obvious that x cannot reoccur in the left part of u of length $\mathrm{per}(u)$. It is then sufficient to scan the right part of u of length $|u| - \mathrm{per}(u) < |u|/2$ to find a non-defined transition in the automaton.

The function disp is implemented directly in the automaton $\mathcal{A}$ without changing the complexity of its construction.

The TRF is presented in Figure 17.

The TRF performs at most $2n$ inspections of text characters and it is also optimal on the average.[10,23]

Summary

The known time complexities of the different algorithms are shown in Table I for both the worst case and the average case.

Considering their average case complexities one should expect the brute-force and the Boyer–Moore–Horspool algorithms to be efficient for large alphabets and the Reverse Factor and the Turbo Reverse Factor to be efficient for large patterns.

THE TEXTS

We give experimental results for the number of inspections of text characters and of the running time for the above algorithms for different types of text: binary alphabet, alphabet of size 4, alphabet of size 8, alphabet of size 20, natural language, C code and genome.

Binary alphabet

The alphabet is $\Sigma = \{a,b\}$. The text is composed of 500,000 characters and was randomly built. The distribution is uniform, with 49·97 per cent of a and 50·03 per cent of b. For patterns of lengths between 2 and 7 we search for all of them and for longer patterns we made the search for only 100 of them randomly built.

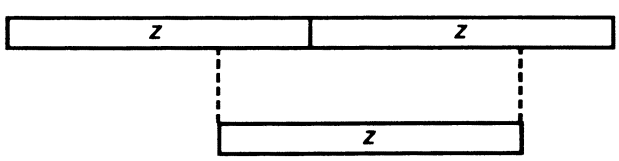

Figure 16. Impossible overlap if z is an acyclic word

```
ALGORITHM TRF;
Begin
    i:=0;
    shift:=m;
    u:=ε;
    while i ≤ n-m do
        j:=m;
        state:=s0;
        u:=m-shift;
        shift:=m;
        while j > u and δ(state,y[i+j]) is defined do
            state:=δ(state,y[i+j]);
            if state ∈ F then
                shift:=j;
            endif
            j:=j-1;
        endwhile
        if j ≤ u then
            if disp(y[i+j+1, i+m], x) = 0 then
                output(i+1);
                shift:=per(x);
            else
                if per(x[1,u]) ≤ u/2 then        /* x[1,u] is periodic */
                    while j > u-per(x[1,u]) and δ(state,y[i+j]) is defined do
                        state:=δ(state,y[i+j]);
                        if state ∈ F then
                            shift:=j;
                        endif
                        j:=j-1;
                    endwhile
                    if j ≤ u then
                        shift:=disp(y[i+j+1, i+m], x);
                    endif
                else                             /* x[1,u] is not periodic */
                    while j > u/2 and δ(state,y[i+j]) is defined do
                        state:=δ(state,y[i+j]);
                        if state ∈ F then
                            shift:=j;
                        endif
                        j:=j-1;
                    endwhile
                endif
            endif
        endif
        i:=i+shift;                  /* shift */
    endwhile
Fin.
```

Figure 17. The Turbo Reverse Factor algorithm

Table I. Time complexities

Algorithms	Worst case	Average case
BF	$O(nm)$	$n(1 + 1/(\|\Sigma\| - 1))$
BM	$O(n + rm)$	
all occurrences		
BM	$3n - n/m$	
first occurrence		
AG	$2n - m + 1$	
TBM	$2n$	
BMH	$O(nm)$	$O(n/\|\Sigma\|)$
QS	$O(nm)$	
RF	$O(nm)$	$O(n(\log_{\|\Sigma\|} m)/m)$
TRF	$2n$	$O(n(\log_{\|\Sigma\|} m)/m)$

Alphabet of size 4

The alphabet is $\Sigma = \{a, b, c, d\}$. The text is composed of 500,000 characters and was randomly built. The distribution is uniform, with 24·97 per cent of a, 25·03 per cent of b, 25·01 per cent of c and 24·99 per cent of d. For patterns of lengths between 2 and 4 we search for all of them and for longer patterns we made the search for only 100 of them randomly built.

Alphabet of size 8

The alphabet is $\Sigma = \{a, b, c, d, e, f, g, h\}$. The text is composed of 500,000 characters and was randomly built. The distribution is uniform, with 12·49 per cent of a, 12·51 per cent of b, 12·5 per cent of c, 12·5 per cent of d, 12·5 per cent of e, 12·52 per cent of f, 12·48 per cent of g and 12·5 per cent of h. For patterns of length 2 and 3 we search for all of them and for longer patterns we made the search for only 100 of them randomly built.

Alphabet of size 20

The alphabet is $\Sigma = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t\}$. The text is composed of 500,000 characters and was randomly built. The distribution is uniform (see Table II). We search all the patterns of length 2 and for longer patterns we made the search for only 100 of them randomly built.

Table II. Character distribution in the text with an alphabet of size 20

Character	Percentage	Character	Percentage	Character	Percentage	Character	Percentage
a	5·03	f	5·06	k	4·97	p	5·01
b	5·01	g	4·97	l	5	q	5·01
c	5·04	h	4·99	m	4·95	r	4·99
d	5·04	i	4·95	n	4·97	s	4·99
e	5·03	j	4·99	o	5·01	t	4·98

Natural language

We use the novel *Alice's Adventures in Wonderland* by Lewis Carroll. The alphabet is composed of 70 different characters (see distribution in Table III). The text is composed of 148,188 characters. For each pattern length we made the search with 100 patterns randomly chosen in the text.

C code

We use a program written in C. The alphabet is composed of 93 different characters (see distribution in Table IV). The text length is equal to 140,240. For each pattern length we search for 100 patterns randomly chosen in the text.

Genome

A genome is a DNA sequence composed of the four nucleotides: Adenine, Cytosine, Guanine and Thymine. The alphabet is $\Sigma = \{a,c,g,t\}$. The genome we used for these tests is a sequence of 180,136 base pairs of the region of the replication origin of the *Bacillus subtilis* chromosome. It is composed of 29·63 per cent of Adenine, 20·56 per cent of Cytosine, 22·95 per cent of Guanine and 26·86 per cent of Thymine. For each pattern length we made the search with 100 of them randomly chosen in the genome.

INSPECTIONS OF TEXT CHARACTERS

For each run of these algorithms we count the number c of inspections for each text character.

Table III. Character distribution in the text in natural language

Character	Percentage	Character	Percentage	Character	Percentage	Character	Percentage
CR	0·7196	E	0·0370	W	0·0474	k	0·2148
space	5·7686	F	0·0146	X	0·0008	l	0·9220
!	0·0892	G	0·0164	Y	0·0226	m	0·3810
″	0·0228	H	0·0560	Z	0·0002	n	1·3766
'	0·3514	I	0·1458	[	0·0004	o	1·5922
(	0·0112	J	0·0016	]	0·0004	p	0·2912
)	0·0110	K	0·0164	_	0·0008	q	0·0250
*	0·0120	L	0·0190	`	0·2198	r	1·0558
,	0·4826	M	0·0388	a	1·6270	s	1·2536
-	0·1338	N	0·0234	b	0·2762	t	2·0394
.	0·1946	O	0·0350	c	0·4490	u	0·6790
:	0·0466	P	0·0128	d	0·9462	v	0·1606
;	0·0386	Q	0·0168	e	2·6716	w	0·4866
?	0·0406	R	0·0278	f	0·3852	x	0·0288
A	0·1276	S	0·0436	g	0·4884	y	0·4298
B	0·0182	T	0·0936	h	1·4148	z	0·0154
C	0·0286	U	0·0126	i	1·3528		
D	0·0380	V	0·0084	j	0·0276		

Table IV. Character distribution in the C program

Character	Percentage	Character	Percentage	Character	Percentage	Character	Percentage
TAB	0·0570	7	0·0190	P	0·0538	i	0·9962
CR	0·9894	8	0·0232	Q	0·0002	j	0·1196
space	5·2162	9	0·0492	R	0·0372	k	0·1798
!	0·0294	:	0·0330	S	0·0742	l	0·5670
″	0·0284	;	0·5474	T	0·4540	m	0·5800
#	0·0134	<	0·0772	U	0·0322	n	0·7376
%	0·0090	=	0·3264	V	0·0328	o	0·3896
&	0·0358	>	0·0268	W	0·0012	p	0·2116
'	0·0014	?	0·0046	X	0·0278	q	0·0190
(	0·6686	A	0·0582	Y	0·1196	r	0·8060
)	0·6688	B	0·0956	Z	0·0236	s	0·3944
*	0·8488	C	0·0132	[	0·1534	u	0·6586
+	0·2220	D	0·0716	\	0·0034	v	0·2820
,	0·4318	E	0·1254	]	0·1534	w	0·0502
-	3·2692	F	0·0252		0·0038	x	0·0938
.	0·1072	G	0·0164	_	0·5248	y	0·0836
/	0·2068	H	0·0072	a	0·7510	z	0·0662
0	0·2034	I	0·2832	b	0·2266	{	0·3854
1	0·1738	J	0·0026	c	0·2610	|	0·0784
2	0·0660	K	0·0344	d	0·4564	}	0·0032
3	0·0338	L	0·2472	e	1·2620		0·0784
4	0·0138	M	0·1160	f	0·3876		
5	0·0108	N	0·2910	g	0·1914		
6	0·0082	O	0·0258	h	0·2032		

Binary alphabet

The BF algorithm scans each text character between 1·5 and 2 times. As BMH and QS algorithms only use the occurrence shift they are not efficient for very small alphabets. The other algorithms are sublinear as expected. For a binary alphabet the best results are achieved by the TRF algorithm; it almost reaches a bound of 2 per cent for very long patterns (see Table V and Figures 18 and 19).

Alphabet of size 4

For very small patterns (length 2 and 3) the QS algorithm is efficient. For longer patterns BM-like algorithms (BM, AG, TBM) and RF-like algorithms (RF, TRF) have better results. For very long patterns RF-like algorithms are the most efficient (see Table VI and Figures 20 and 21).

Alphabet of size 8

For small patterns (up to length 7) the QS algorithm is the most efficient. For $m = 8$, 9 and 10 BM-like algorithms become efficient, and for longer patterns RF-like algorithms are the most efficient. The TRF algorithm even reaches a bound of less than 1 per cent for very long patterns (see Table VII and Figures 22 and 23).

Table V. Text character inspections for a binary alphabet

m	BF	BM	AG	TBM	BMH	QS	RF	TRF
2	1·4999	1·0001	**0·9166**	1·0001	1·0001	0·9482	1·4168	1·0001
3	1·7499	0·9731	**0·8969**	0·9538	1·0945	1·0562	1·2400	0·9315
4	1·8749	0·9227	**0·8567**	0·8830	1·1659	1·0915	1·0612	0·8659
5	1·9374	0·8576	0·8045	0·8216	1·2099	1·0994	0·9095	**0·7914**
6	1·9687	0·7992	0·7532	0·7624	1·2338	1·0990	0·7888	**0·7205**
7	1·9843	0·7446	0·7041	0·7108	1·2467	1·0967	0·6948	**0·6563**
8	1·9921	0·6997	0·6599	0·6733	1·2526	1·0496	0·6214	**0·6003**
9	1·9960	0·6549	0·6209	0·6263	1·2398	1·0765	0·5569	**0·5450**
10	1·9980	0·6121	0·5815	0·5851	1·2139	1·0929	0·5127	**0·5064**
20	1·9999	0·4505	0·4280	0·4350	1·2680	1·1130	0·2942	**0·2941**
40	2·0001	0·3291	0·3135	0·3220	1·2320	1·0951	0·1696	**0·1695**
80	2·0000	0·2700	0·2560	0·2633	1·2547	1·0824	0·0970	**0·0967**
160	2·0006	0·2104	0·2002	0·2071	1·2272	1·0893	0·0560	**0·0554**
320	2·0011	0·1815	0·1724	0·1803	1·2466	1·0859	0·0338	**0·0326**
640	2·0019	0·1598	0·1513	0·1574	1·2835	1·1057	0·0238	**0·0211**

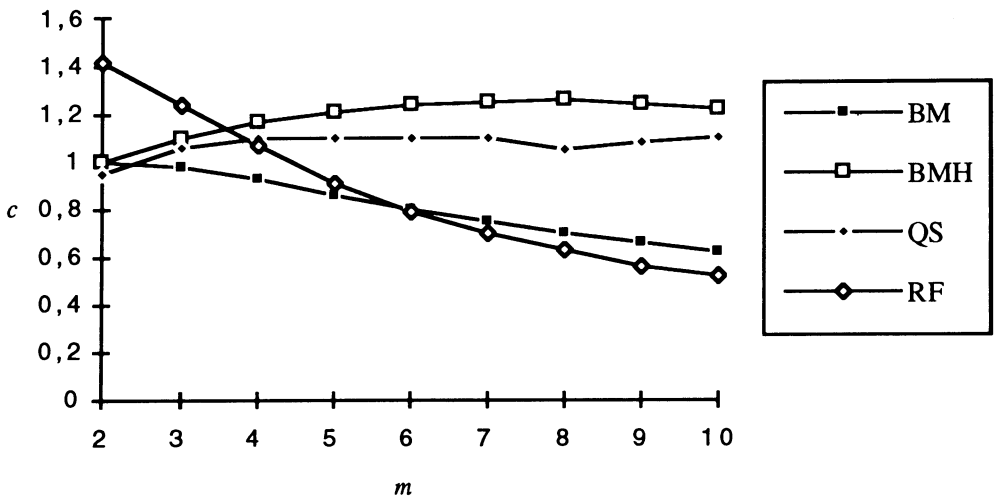

Figure 18. Text character inspections for a binary alphabet—short patterns

Alphabet of size 20

For small patterns the QS algorithm is the most efficient. Then for $m = 20$ BM-like algorithms are the best. For longer patterns RF-like algorithms are the most efficient (see Table VIII and Figures 24 and 25).

Text in natural language

For $m \leq 10$ the QS algorithm is the most efficient. This case corresponds to the typical search for words in a text editor for instance. For longer patterns RF-like algorithms are better (see Table IX and Figures 26 and 27).

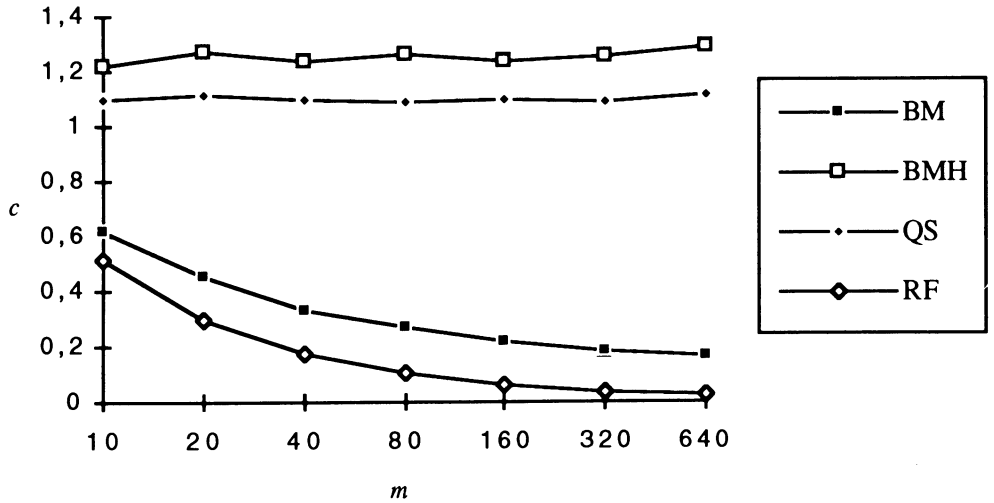

Figure 19. Text character inspections for a binary alphabet—long patterns

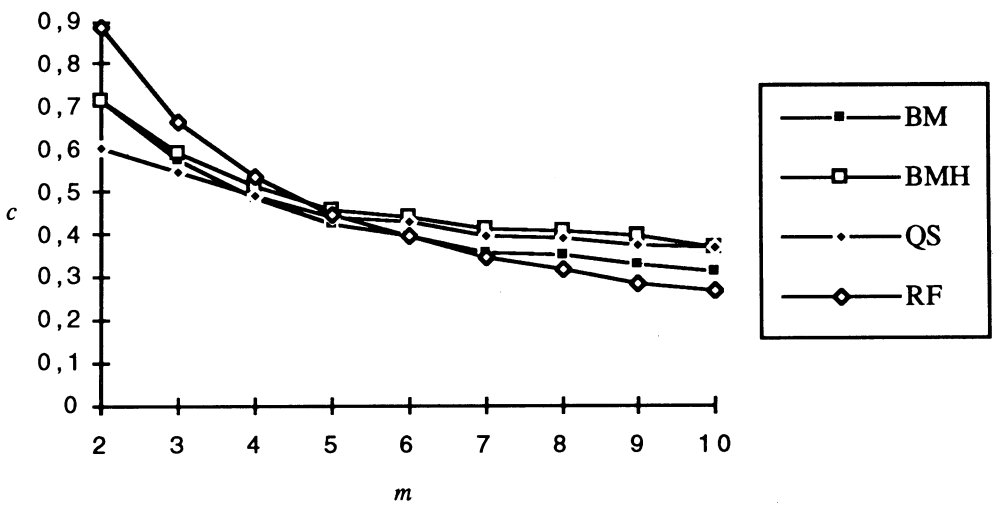

Figure 20. Text character inspections for an alphabet of size 4—short patterns

C code

For small patterns ($m \leq 4$) the QS algorithm is the most efficient. Then for pattern lengths between 5 and 20 the AG algorithm is the best. This is due to the great amount of repetitions which are present in a C program. Then for very long patterns ($m \geq 40$) the TRF algorithm is the best (see Table X and Figures 28 and 29).

Genome

The results are very similar to those for an alphabet of size 4 though the distribution of the four nucleotides is not uniform, except that this time BM-like algorithms are efficient up to pattern length 6 (see Table XI and Figures 30 and 31).

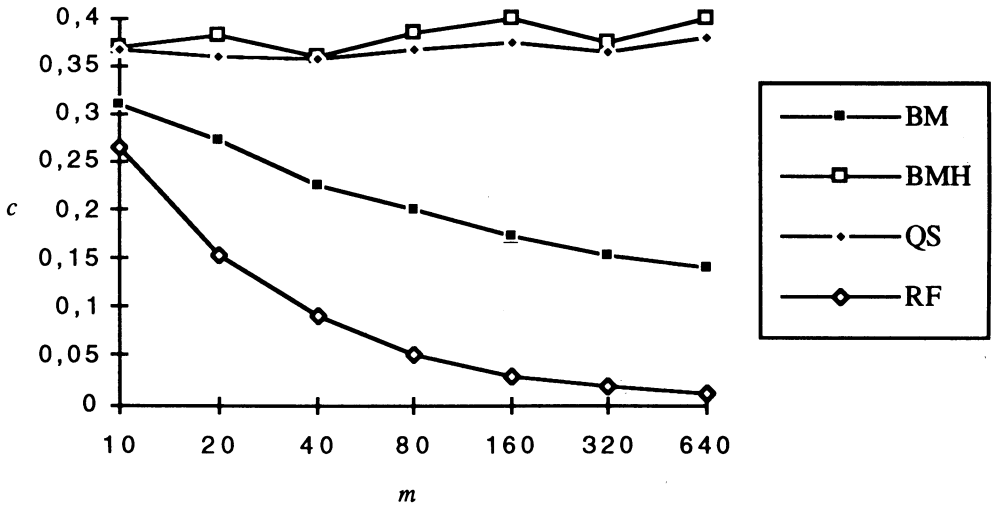

Figure 21. Text character inspections for an alphabet of size 4—long patterns

Table VI. Text character inspections for an alphabet of size 4

m	BF	BM	AG	TBM	BMH	QS	RF	TRF
2	1·2499	0·7142	0·7053	0·7142	0·7142	**0·6038**	0·8839	0·7857
3	1·3124	0·5716	0·5675	0·5711	0·5892	**0·5454**	0·6631	0·6287
4	1·3281	0·4845	**0·4821**	0·4839	0·5136	0·4894	0·5320	0·5213
5	1·3319	0·4238	**0·4220**	0·4233	0·4581	0·4416	0·4464	0·4432
6	1·3329	0·3960	0·3945	0·3954	0·4382	0·4269	0·3928	**0·3919**
7	1·3332	0·3579	0·3564	0·3573	0·4091	0·3959	0·3447	**0·3445**
8	1·3333	0·3493	0·3478	0·3488	0·4051	0·3902	0·3152	**0·3151**
9	1·3332	0·3287	0·3272	0·3279	0·3961	0·3731	0·2851	**0·2850**
10	1·3332	0·3084	0·3073	0·3079	0·3681	0·3659	**0·2653**	**0·2653**
20	1·3332	0·2713	0·2701	0·2709	0·3809	0·3588	0·1542	**0·1541**
40	1·3333	0·2257	0·2247	0·2254	0·3596	0·3562	0·0899	**0·0898**
80	1·3334	0·2011	0·2001	0·2016	0·3848	0·3667	0·0514	**0·0513**
160	1·3335	0·1736	0·1729	0·1748	0·3994	0·3750	0·0295	**0·0292**
320	1·3337	0·1542	0·1535	0·1553	0·3740	0·3650	0·0179	**0·0171**
640	1·3341	0·1400	0·1394	0·1414	0·3987	0·3800	0·0125	**0·0112**

We can make some remarks on these tests regarding the number of inspections of text characters. It seems that BMH algorithm is not very efficient from this point of view.

For long patterns

1. The difference between the number of text character inspections performed by the BM algorithm and the number of text characer inspections performed by the AG algorithm decreases: this means that the AG algorithm then performs few jumps (except for C code).
2. The difference between the number of text character inspections performed by the RF algorithm and the number of text character inspections performed by

Table VII. Text character inspections for an alphabet of size 8

m	BF	BM	AG	TBM	BMG	QS	RF	TRF
2	1·1249	0·6000	0·5989	0·6000	0·6000	**0·4625**	0·6740	0·6501
3	1·1406	0·4341	0·4339	0·4341	0·4371	**0·3771**	0·4808	0·4767
4	1·1425	0·3453	0·3452	0·3453	0·3503	**0·3177**	0·3810	0·3804
5	1·1428	0·2948	0·2948	0·2948	0·2998	**0·2811**	0·3236	0·3235
6	1·1428	0·2603	0·2602	0·2603	0·2665	**0·2560**	0·2832	0·2832
7	1·1428	**0·2344**	**0·2344**	**0·2344**	0·2414	**0·2344**	0·2516	0·2516
8	1·1428	**0·2167**	**0·2167**	**0·2167**	0·2236	0·2194	0·2272	0·2272
9	1·1428	**0·2025**	**0·2025**	**0·2025**	0·2105	0·2076	0·2091	0·2091
10	1·1428	0·1911	**0·1910**	0·1911	0·2007	0·1966	0·1929	0·1929
20	1·1428	0·1519	0·1519	0·1519	0·1652	0·1637	0·1132	**0·1131**
40	1·1428	0·1305	0·1304	0·1305	0·1494	0·1478	0·0638	**0·0637**
80	1·1428	0·1271	0·1271	0·1272	0·1524	0·1511	0·0361	**0·0360**
160	1·1427	0·1169	0·1168	0·1170	0·1544	0·1492	0·0205	**0·0204**
320	1·1427	0·1102	0·1102	0·1104	0·1564	0·1522	0·0121	**0·0117**
640	1·1426	0·1018	0·1018	0·1020	0·1524	0·1491	0·0079	**0·0073**

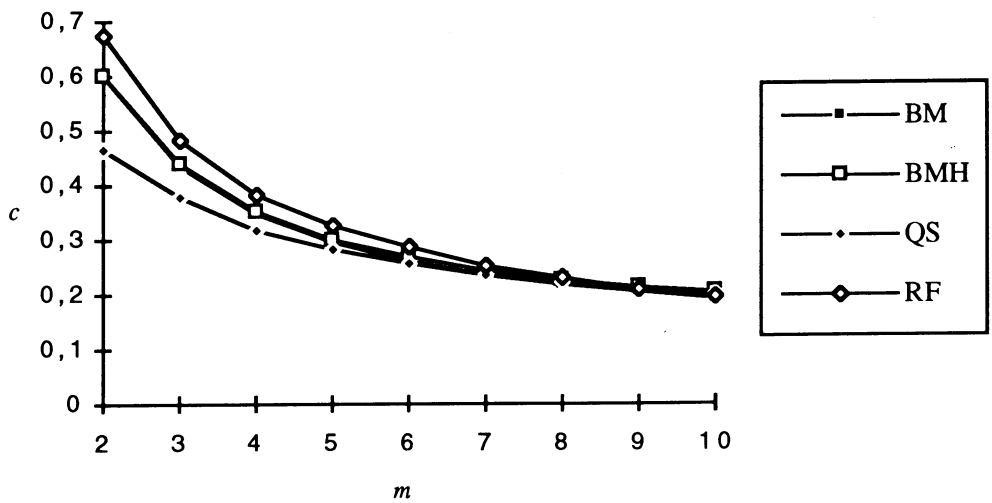

Figure 22. Text character inspections for an alphabet of size 8—short patterns

the TRF algorithm decreases: this means that TRF algorithm scarcely reaches the decision point (except for C code).

For long patterns or for small alphabets the TRF algorithm is always the best. For the other cases the QS algorithm is efficient.

However, all these algorithms do not make the same use of text characters: for BMH, BM, AG, TBM and QS text characters are used to perform comparisons with pattern characters, for the RF and TRF algorithms text characters are used to perform transitions in an automaton.

Furthermore, we have not taken into account the fact that for computing its shifts the QS algorithm inspects a text character that has not yet been inspected.

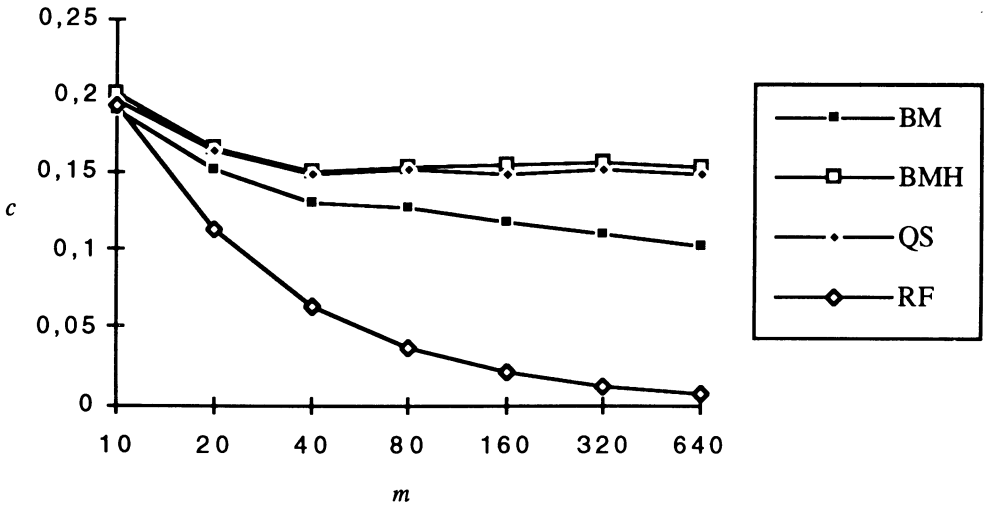

Figure 23. Text character inspections for an alphabet of size 8—long patterns

Table VIII. Text character inspections for an alphabet of size 20

m	BF	BM	AG	TBM	BMH	QS	RF	TRF
2	1·0499	0·5384	0·5383	0·5384	0·5384	**0·3839**	0·5653	0·5615
3	1·0524	0·3696	0·3696	0·3696	0·3700	**0·2973**	0·3908	0·3905
4	1·0526	0·2837	0·2837	0·2837	0·2841	**0·2422**	0·3020	0·3020
5	1·0525	0·2324	0·2324	0·2324	0·2331	**0·2069**	0·2500	0·2500
6	1·0526	0·1988	0·1988	0·1988	0·1993	**0·1820**	0·2158	0·2158
7	1·0526	0·1745	0·1745	0·1745	0·1752	**0·1626**	0·1903	0·1903
8	1·0526	0·1563	0·1563	0·1563	0·1570	**0·1477**	0·1709	0·1709
9	1·0526	0·1420	0·1420	0·1420	0·1430	**0·1359**	0·1556	0·1556
10	1·0526	0·1308	0·1308	0·1308	0·1318	**0·1263**	0·1436	0·1436
20	1·0526	**0·0813**	**0·0813**	**0·0813**	0·0822	0·0817	0·0850	0·0850
40	1·0525	0·0594	0·0594	0·0594	0·0610	0·0608	**0·0494**	**0·0494**
80	1·0524	0·0530	0·0530	0·0530	0·0549	0·0548	**0·0271**	**0·0271**
160	1·0522	0·0513	0·0513	0·0513	0·0537	0·0542	**0·0146**	**0·0146**
320	1·0519	0·0504	0·0504	0·0504	0·0542	0·0541	**0·0080**	**0·0080**
640	1·0512	0·0494	0·0494	0·0494	0·0546	0·0542	**0·0044**	**0·0044**

RUNNING TIMES

Though the measure of text character inspections is an objective parameter of the performance of string matching algorithms, one can expect that the best theoretical string matching algorithms have good running times. In order to evaluate the practical performances of string matching algorithms, we had implemented them in C in a homogeneous way to make the comparison significant.

For each of them we count the running time of both the preprocessing phase and the searching phase for one pattern. The running times are expressed in hundredth of seconds.

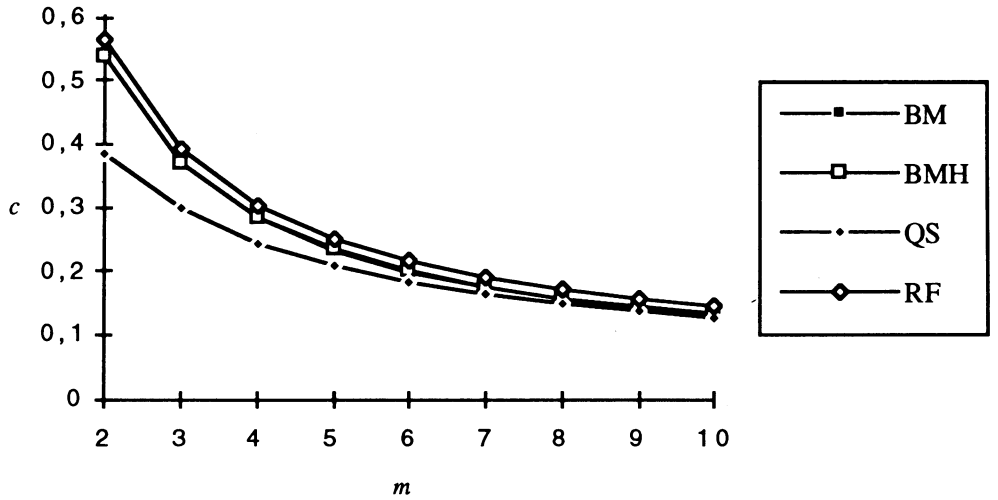

Figure 24. Text character inspections for an alphabet of size 20—short patterns

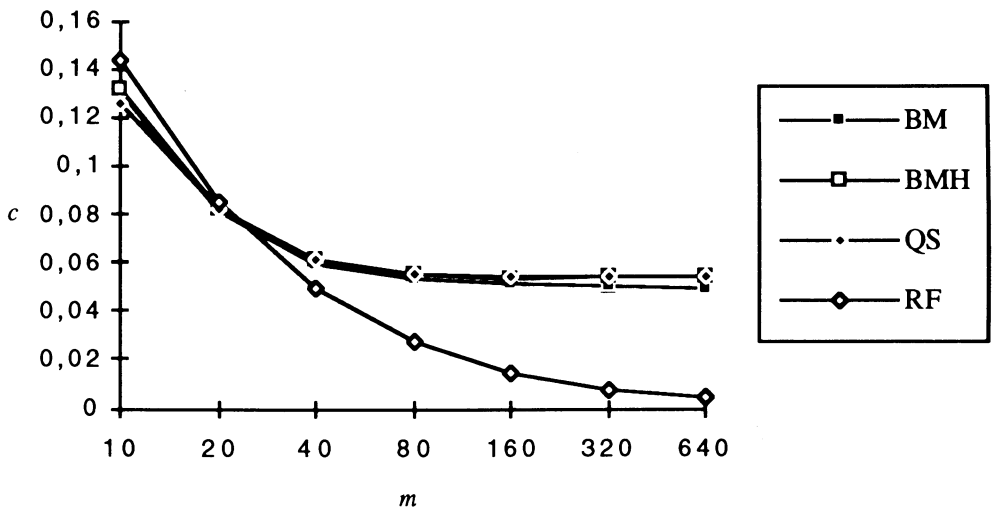

Figure 25. Text character inspections for an alphabet of size 20—long patterns

For the RF and TRF algorithms, the suffix automaton has been built in a table in $O(|\Sigma|m)$ space.

Binary alphabet

We can make the same remarks as in the measure of the number of text character inspections: BMH and QS algorithms are not efficient for a very small alphabet as they only use the occurrence shift. The BF algorithm is linear.

The QS algorithm is efficient for very short patterns ($m = 2$ and $m = 3$) then for pattern lengths up to 6 BM is the best, and after RF is the most efficient (see

Table IX. Text character inspections for an English text

m	BF	BM	AG	TBM	BMH	QS	RF	TRF
2	1·0763	0·5635	0·5626	0·5635	0·5635	**0·4115**	0·6081	0·5963
3	1·0838	0·3915	0·3905	0·3910	0·3943	**0·3219**	0·4210	0·4170
4	1·0855	0·3055	0·3042	0·3047	0·3077	**0·2687**	0·3293	0·3265
5	1·0863	0·2567	0·2552	0·2556	0·2593	**0·2330**	0·2756	0·2731
6	1·0869	0·2221	0·2204	0·2208	0·2254	**0·2055**	0·2385	0·2360
7	1·0874	0·1956	0·1938	0·1942	0·1990	**0·1842**	0·2102	0·2079
8	1·0878	0·1759	0·1746	0·1749	0·1804	**0·1689**	0·1893	0·1873
9	1·0880	0·1599	0·1593	0·1594	0·1636	**0·1556**	0·1717	0·1702
10	1·0881	0·1487	0·1480	0·1481	0·1533	**0·1459**	0·1578	0·1567
20	1·0886	0·0892	**0·0887**	0·0888	0·0947	0·0909	0·0902	0·0896
40	1·0886	0·0600	0·0599	0·0599	0·0651	0·0633	0·0528	**0·0526**
80	1·0886	0·0424	0·0424	0·0424	0·0463	0·0455	0·0311	**0·0308**
160	1·0885	0·0338	0·0337	0·0338	0·0378	0·0362	0·0189	**0·0184**
320	1·0884	0·0282	0·0282	0·0282	0·0311	0·0309	0·0131	**0·0121**
640	1·0883	0·0253	0·0253	0·0253	0·0284	0·0279	0·0120	**0·0098**

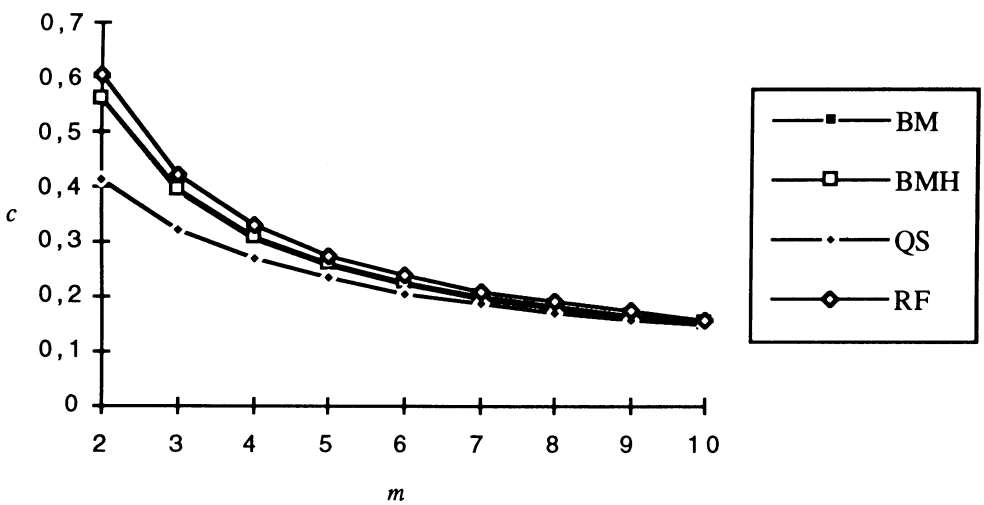

Figure 26. Text character inspections for an English text—short patterns

Table XII and Figures 32 and 33). For very long patterns, the running times for RF and TRF algorithms increase because of the preprocessing phase, the time for which is equal to one third of the time for the searching phase.

Alphabet of size 4

For very short patterns ($m \leq 4$) the QS algorithm is the best. Then the RF algorithm is better (see Table XIII and Figures 34 and 35). For very long patterns, the running times for RF and TRF algorithms increase because of the preprocessing phase, the time for which is equal to half of the time for the searching phase.

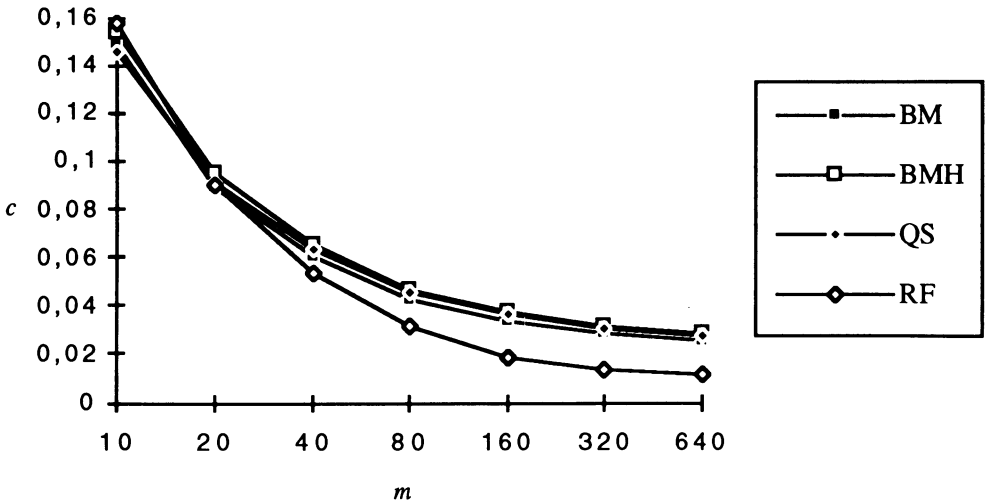

Figure 27. Text character inspections for an English text—long patterns

Table X. Text character inspections for C code

m	BF	BM	AG	TBM	BMH	QS	RF	TRF
2	1·0458	0·5508	0·5356	0·5508	0·5508	**0·3928**	0·5864	0·5527
3	1·0634	0·4001	0·3724	0·3863	0·4049	**0·3208**	0·4250	0·3806
4	1·0783	0·3270	0·2914	0·3034	0·3333	**0·2864**	0·3528	0·3004
5	1·0909	0·2910	**0·2441**	0·2561	0·3065	0·2622	0·3099	0·2493
6	1·1031	0·2669	**0·2099**	0·2215	0·2827	0·2482	0·2840	0·2147
7	1·1147	0·2471	**0·1857**	0·1962	0·2663	0·2344	0·2681	0·1909
8	1·1251	0·2359	**0·1655**	0·1758	0·2558	0·2292	0·2541	0·1713
9	1·1352	0·2316	**0·1521**	0·1623	0·2559	0·2233	0·2468	0·1565
10	1·1452	0·2253	**0·1377**	0·1476	0·2435	0·2193	0·2418	0·1439
20	1·2322	0·2094	**0·0843**	0·0910	0·2484	0·2177	0·2360	0·0879
40	1·3392	0·1973	0·0549	0·0587	0·3085	0·2054	0·2038	**0·0541**
80	1·3881	0·0360	0·0349	0·0352	0·2129	0·0487	0·0358	**0·0345**
160	1·3883	0·0253	0·0236	0·0243	0·1819	0·0413	0·0225	**0·0219**
320	1·3883	0·0168	0·0164	0·0165	0·1042	0·0270	0·0167	**0·0156**
640	1·3882	0·0161	0·0153	0·0157	0·1062	0·0210	0·0158	**0·0135**

Alphabet of size 8

For pattern lengths up to 4 the QS algorithm is the best; for longer patterns the RF algorithm is the best (see Table XIV and Figures 36 and 37). These results are almost identical to those for an alphabet of size 4, except that the BMH and QS algorithms are more efficient in this case. For very long patterns, the running times for the RF and TRF algorithms increase because of the preprocessing phase, the time for which is equal to half of the time for the searching phase.

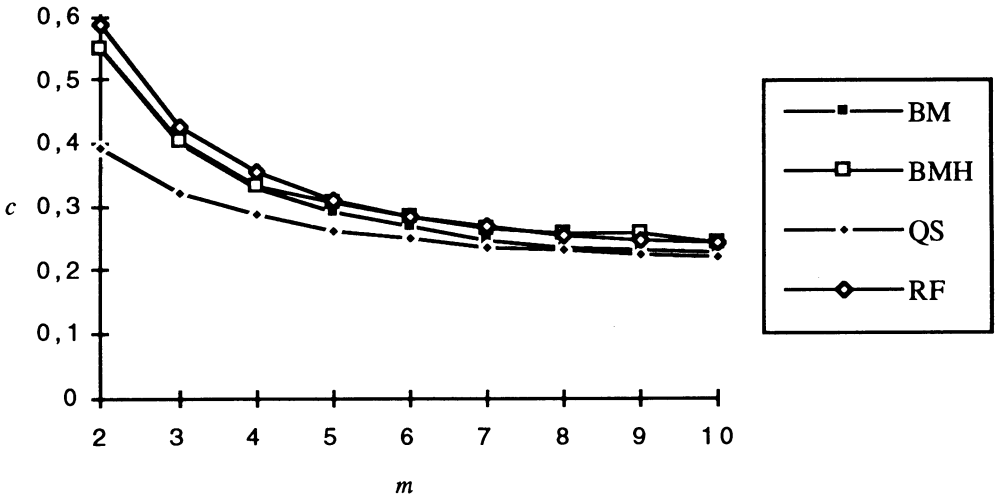

Figure 28. Text character inspections for C code—short patterns

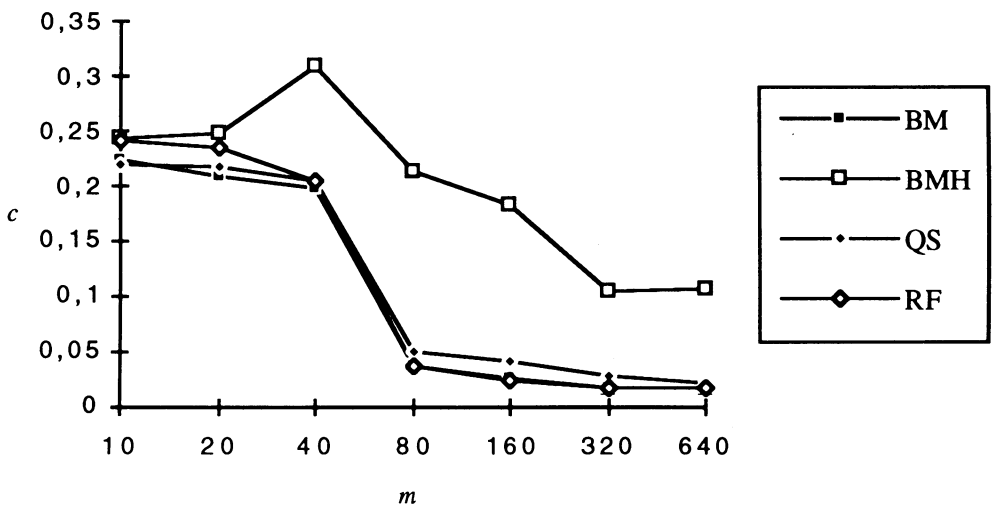

Figure 29. Text character inspections for C code—long patterns

Alphabet of size 20

For $m < 4$ the QS algorithm is the best algorithm; then for $m \geq 4$ the RF algorithm is the quickest (see Table XV and Figures 38 and 39). For very long patterns, the running times for RF and TRF algorithms increase because of the preprocessing phase, the time for which is equal to the time for the searching phase.

Text in natural language

The QS algorithm is the best for pattern lengths up to 7, which is generally the typical search in such a text. Then, for long patterns the RF algorithm is quicker,

Table XI. Text character inspections for a genome

m	BF	BM	AG	TBM	BMH	QS	RF	TRF
2	1·2538	0·7237	0·7069	0·7237	0·7237	**0·5992**	0·8917	0·7836
3	1·3211	0·5717	0·5652	0·5700	0·5927	**0·5511**	0·6689	0·6267
4	1·3391	0·4947	**0·4918**	0·4940	0·5221	0·5037	0·5450	0·5311
5	1·3438	0·4336	**0·4305**	0·4327	0·4888	0·4544	0·4544	0·4499
6	1·2351	0·3888	**0·3863**	0·3880	0·4503	0·4218	0·3908	0·3893
7	1·3455	0·3591	0·3572	0·3583	0·4189	0·3959	0·3478	**0·3473**
8	1·3455	0·3376	0·3356	0·3368	0·4065	0·3830	0·3129	**0·3128**
9	1·3456	0·3255	0·3237	0·3248	0·3960	0·3735	0·2863	**0·2862**
10	1·3456	0·3150	0·3134	0·3143	0·3866	0·3628	**0·2643**	**0·2643**
20	1·3456	0·2657	0·2646	0·2649	0·3762	0·3767	**0·1560**	**0·1560**
40	1·3455	0·2322	0·2313	0·2317	0·3721	0·3569	0·0903	**0·0902**
80	1·3454	0·2024	0·2013	0·2026	0·3761	0·3518	0·0518	**0·0516**
160	1·3453	0·1735	0·1727	0·1743	0·3661	0·3567	0·0300	**0·0295**
320	1·3450	0·1574	0·1564	0·1585	0·3964	0·3593	0·0185	**0·0176**
640	1·3444	0·1485	0·1476	0·1492	0·3754	0·3552	0·0140	**0·0122**

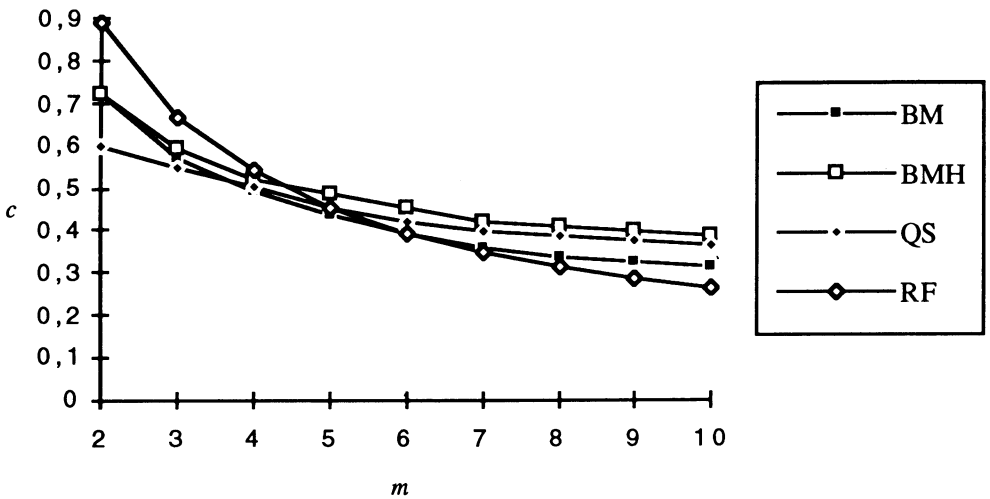

Figure 30. Text character inspections for a genome—short patterns

except for very long patterns, where the time for the preprocessing phase is equal to the time for the searching phase. This is due to the fact that the length of the text is not very great (see Table XVI and Figures 40 and 41).

C code

The QS algorithm is efficient up to pattern length 4, then the RF algorithm is better except for very long patterns where the time for the preprocessing phase equals the time for the searching phase (see Table XVII and Figures 42 and 43).

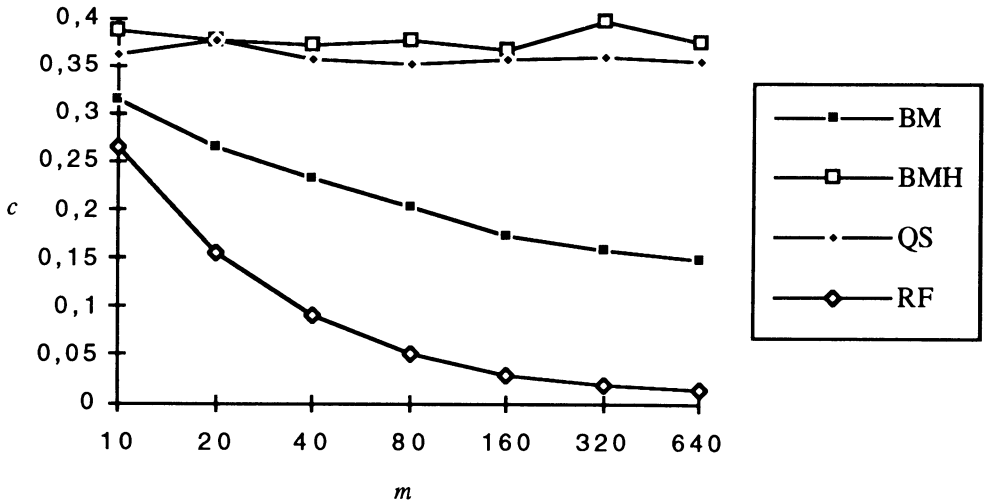

Figure 31. Text character inspections for a genome—long patterns

Table XII. Running times for a binary alphabet

m	BF	BM	AG	TBM	BMH	QS	RF	TRF
2	13·9000	12·2000	29·7000	21·5000	14·4000	**11·8500**	18·3500	28·4500
3	15·1500	12·5500	27·4250	16·9000	13·0250	**11·6750**	16·4500	20·1250
4	16·1750	**10·1250**	24·6625	15·2500	12·4750	11·2500	12·9750	17·5125
5	16·4187	**9·6500**	22·1996	14·4125	12·0562	10·8875	10·8625	15·1375
6	16·0531	**8·6592**	20·9842	13·2875	12·2437	11·1250	9·3343	13·6625
7	16·0108	8·0515	20·0077	12·3484	12·2046	10·8994	**7·9859**	12·1062
8	16·5159	7·5980	19·1420	11·5280	12·2220	10·3700	**7·3059**	11·3460
9	16·4599	7·0859	18·1399	10·6460	11·9620	10·6420	**6·4740**	10·2780
10	16·5460	6·6700	17·3239	9·9719	11·8540	10·9639	**6·0180**	9·6900
20	16·5420	4·9260	13·4699	7·4920	12·2860	11·0639	**3·9820**	6·1420
40	16·5280	3·6620	10·3933	5·6480	11·8960	10·8780	**2·6560**	3·9560
80	16·5280	3·0480	8·4459	4·6540	12·1180	10·7499	**1·8780**	2·7060
160	16·5280	2·3980	6·6340	3·6740	11·8540	10·8700	**1·2340**	2·0120
320	16·5380	2·0380	5·7440	3·1680	12·0360	10·8439	**0·9300**	1·6520
640	16·5399	1·8140	4·9880	2·7600	12·4039	11·0520	**1·2100**	1·8640

Genome

The results for a genome are similar to the results for an alphabet of size 4. Except that in this case the QS algorithm is still the best for $m = 5$, but this is due to the fact that the length of the text is smaller in this case (see Table XVIII and Figures 44 and 45).

It is possible to make some observations on these results.

The BM algorithm is always faster than its refinements (the AG and TBM algorithms). It is the same with the RF algorithm, which is always better than the TRF algorithm. This means that the work done in order to save comparisons has a cost too high to be efficient.

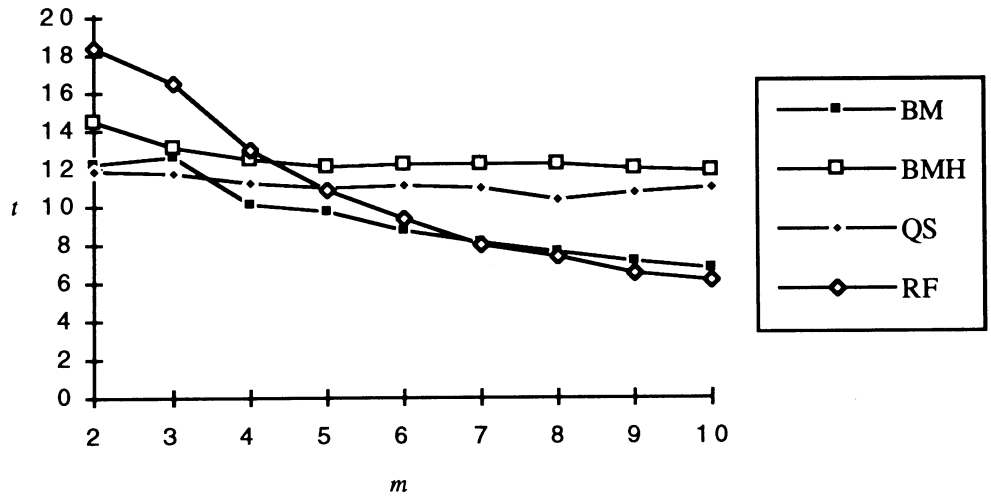

Figure 32. Running times for a binary alphabet—short patterns

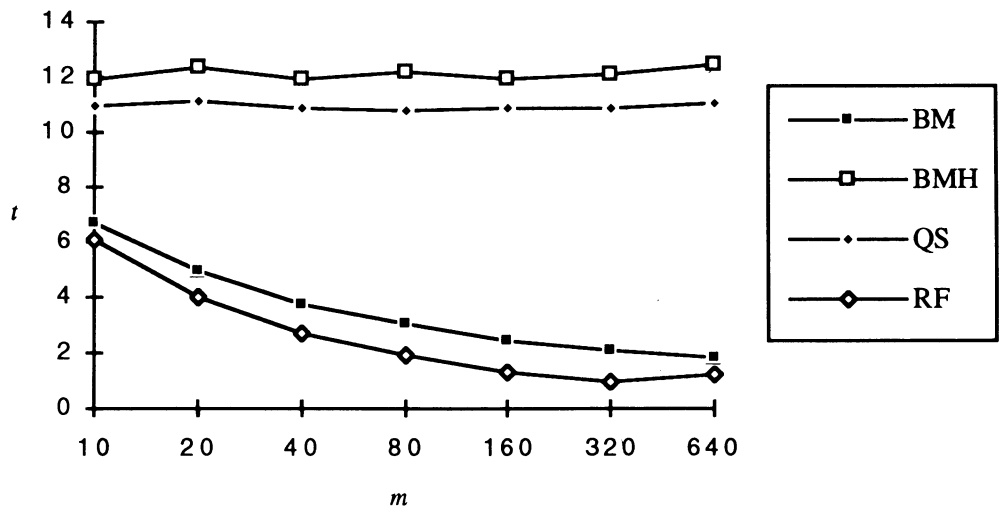

Figure 33. Running times for a binary alphabet—long patterns

The QS algorithm proves to be a very good algorithm in practice for large alphabets and short patterns. Then it is typically suited for search in a natural language context.

The RF algorithm is efficient for long patterns and small alphabets. In this case it is not very annoying to implement the suffix automaton in a table. This can be the case for search in a genome where the alphabet is composed of only four characters.

acknowledgements

I am indebted to the referees for their useful comments.

Table XIII. Running times for an alphabet of size 4

m	BF	BM	AG	TBM	BMH	QS	RF	TRF
2	12·1000	11·1125	23·5125	17·1500	10·5875	**7·5875**	11·1500	16·8000
3	12·3093	8·3656	18·8406	13·5218	7·7750	**6·5000**	7·6218	12·2781
4	12·2062	6·9562	16·2179	11·3151	6·6328	**5·7648**	6·0007	9·6296
5	13·1160	6·0780	14·8160	9·7439	6·0320	5·3799	**5·1660**	8·1120
6	13·1080	5·7059	14·2500	9·0700	5·6220	5·2220	**4·5619**	7·2000
7	13·1360	5·1420	13·4320	8·2479	5·2680	4·8740	**4·0080**	6·4200
8	13·1240	5·0740	13·2880	8·0440	5·2120	4·8600	**3·7020**	5·9200
9	13·0980	4·7720	12·7860	7·5800	5·1100	4·6700	**3·3780**	5·4520
10	13·1179	4·4900	12·2779	7·1020	4·7500	4·6100	**3·1720**	5·1399
20	13·1200	3·9960	10·9679	6·2840	4·9200	4·5720	**2·1840**	3·2920
40	13·1160	3·3620	9·5199	5·2679	4·6480	4·5460	**1·5720**	2·1780
80	13·1080	3·0040	8·3739	4·7160	4·9560	4·6640	**1·1380**	1·5300
160	13·1140	2·5880	7·2280	4·0880	5·1419	4·7580	**0·6380**	1·1360
320	13·1159	2·3160	6·4640	3·6480	4·8160	4·6659	**0·5620**	0·9520
640	13·1040	2·1060	5·7720	3·3199	5·1120	4·8240	**0·8300**	1·1280

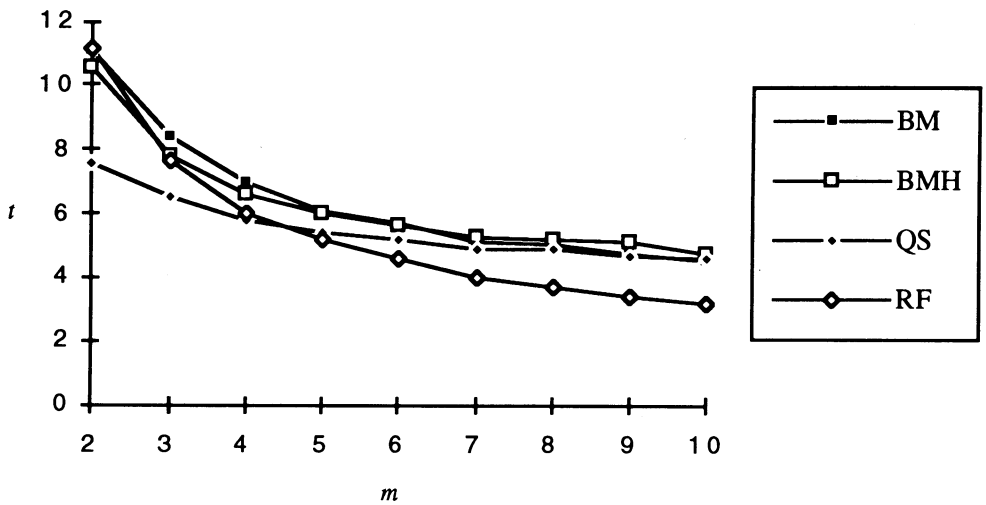

Figure 34. Running times for an alphabet of size 4—short patterns

REFERENCES

1. R.L. Rivest, 'On the worst case behavior of string searching algorithms', *SIAM J. Comput.*, **6**, 669–674 (1977).
2. M. Crochemore, 'String-matching and periods', *Theoret. Comput. Sci.*, **92**, 33–47 (1992).
3. Z. Galil and J. Seiferas, 'Time-space optimal string matching', *J. Comput. Syst. Sci.*, **26**, 280–294, (1983).
4. M. Crochemore and D. Perrin, 'Two-way string-matching', *J. ACM*, **38**, 651–675 (1991).
5. J.H. Morris Jr. and V.R. Pratt, 'A linear pattern-matching algorithm', *Report 40*, University of California, Berkeley, 1970.
6. D.E. Knuth, J.H. Morris Jr. and V.R. Pratt, 'Fast pattern matching in strings', *SIAM J. Comput.*, **6**, 323–350 (1977).
7. C. Hancart, 'On Simon's string searching algorithm', *Inf. Process. Lett.*, **47**, 95–99 (1993).
8. D.M. Sunday, 'A very fast substring search algorithm', *Comm. ACM*, **33**, 132–142 (1990).

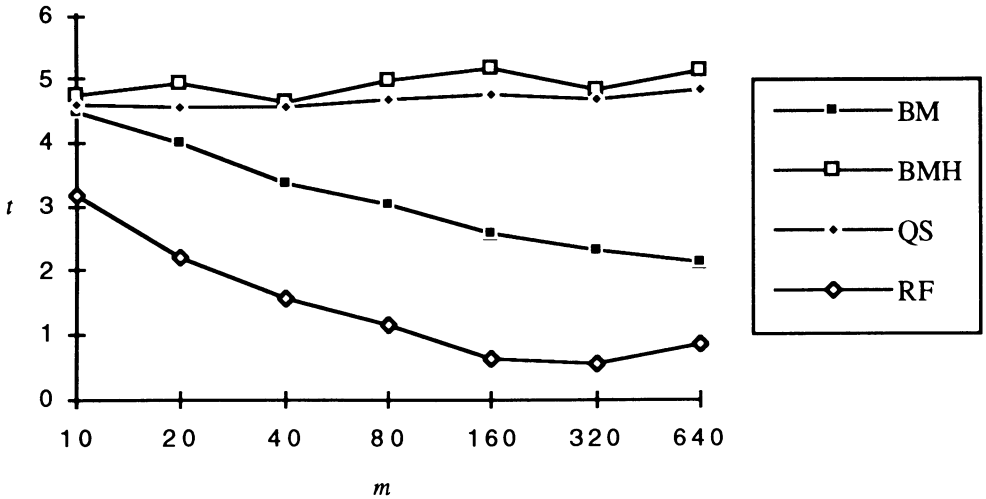

Figure 35. Running times for an alphabet of size 4—long patterns

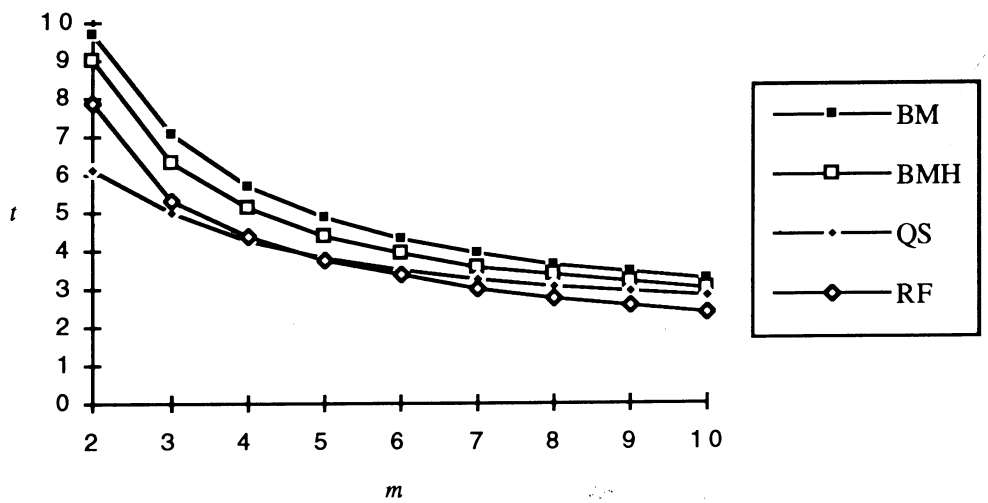

Figure 36. Running times for an alphabet of size 8—short patterns

9. R.S. Boyer and J.S. Moore, 'A fast string searching algorithm', *Comm. ACM*, **20**, 762–772 (1977).
10. Z. Galil, 'On improving the worst case running time of the Boyer–Moore string searching algorithm', *Comm. ACM*, **22**, 505–508 (1979).
11. A. Apostolico and R. Giancarlo, 'The Boyer–Moore–Galil string searching stategies revisited', *SIAM J. Comput.*, **15**, 98–105 (1986).
12. M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski and W. Rytter, 'Speeding-up two string-matching algorithms', in A. Finkel and M. Jantzen (eds), *9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, Lecture Notes in Computer Science 577*, Springer-Verlag, Berlin, 1992, pp. 589–600.
13. R.N. Horspool, 'Practical fast searching in string', *Software–Practice and Experience*, **10**, 501–506 (1980).
14. T. Lecroq, 'A variation on the Boyer–Moore algorithm', *Theoret. Comput. Sci.*, **92**, 119–144 (1992).
15. L. Colussi, 'Correctness and efficiency of the pattern matching algorithms', *Information and Computation*, **95**, 225–251 (1991).

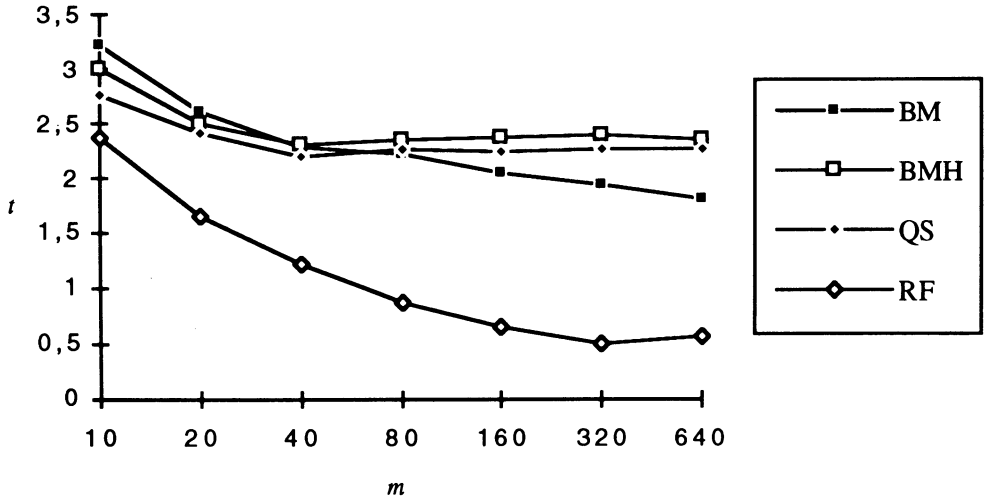

Figure 37. Running times for an alphabet of size 8—long patterns

Table XIV. Running times for an alphabet of size 8

m	BF	BM	AG	TBM	BMH	QS	RF	TRF
2	12·2062	9·6406	21·2062	16·0123	8·9967	**6·1062**	7·8687	13·0279
3	12·2062	7·0273	15·9449	11·5695	6·3042	**4·9730**	5·2855	9·0226
4	12·2019	5·6500	13·3520	9·2060	5·1160	**4·2440**	4·3540	7·2160
5	12·2099	4·8400	12·2700	7·8860	4·3740	3·8080	**3·7460**	6·1880
6	12·1940	4·2940	11·6140	7·0000	3·9260	3·4940	**3·3340**	5·4160
7	12·1720	3·8940	11·1220	6·3240	3·5520	3·2160	**3·0000**	4·8340
8	12·1940	3·6100	10·8239	5·8660	3·3220	3·0560	**2·7440**	4·4460
9	12·2159	3·3900	10·3980	5·4980	3·1440	2·9240	**2·5439**	4·0860
10	12·1959	3·2220	9·9639	5·2160	3·0020	2·7760	**2·3760**	3·8220
20	12·1539	2·6180	8·4100	4·1840	2·5060	2·4160	**1·6420**	2·5060
40	12·1840	2·2920	7·4359	3·6340	2·3040	2·2080	**1·2120**	1·6800
80	12·1939	2·2300	7·2500	3·5180	2·3420	2·2560	**0·8760**	1·1880
160	12·2220	2·0400	6·6920	3·2560	2·3740	2·2380	**0·6520**	0·9240
320	12·1900	1·9320	6·2400	3·0660	2·3880	2·2680	**0·4840**	0·6920
640	12·1680	1·8120	5·7760	2·8640	2·3420	2·2560	**0·5500**	0·7960

16. Z. Galil, R. Giancarlo, 'On the exact complexity of string matching: upper bounds', *SIAM J. Comput.*, **21**, 407–437 (1992).
17. G. Davies and S. Bowler, 'Algorithms for pattern matching', *Software—Practice and Experience*, **16**(6), 575–601 (1986).
18. R.A. Baeza-Yates, 'Improved string searching', *Software—Practice and Experience*, **19**(3), 257–271 (1989).
19. P.D. Smith, 'Experiments with a very fast substring search algorithm', *Software—Practice and Experience*, **21**(10), 1065–1074 (1991).
20. A. Hume and D.M. Sunday, 'Fast string searching', *Software—Practice and Experience*, **21**(11), 1221–1248 (1991).
21. T. Raita, 'Tuning the Boyer–Moore–Horspool string searching algorithm', *Software—Practice and Experience*, **22**(10), 879–884 (1992).

Table XV. Running times for an alphabet of size 20

m	BF	BM	AG	TBM	BMH	QS	RF	TRF
2	10·6635	9·2199	19·8179	15·4434	8·2180	**5·4884**	6·1625	10·9664
3	10·6187	6·2961	14·5018	10·5832	5·6190	**4·1846**	4·2906	7·5394
4	11·7100	4·9600	12·0540	8·1539	4·4040	3·5620	**3·4440**	5·9600
5	11·7040	4·1060	10·9620	6·7280	3·6500	3·0800	**2·9140**	4·9600
6	11·7040	3·5480	10·3760	5·8040	3·1540	2·7540	**2·4140**	4·3200
7	11·7160	3·1460	9·9760	5·1220	2·7960	2·4860	**2·2760**	3·8539
8	11·7560	2·8460	9·6740	4·6160	2·5320	2·2780	**2·0980**	3·5340
9	11·7239	2·6020	9·1560	4·2079	2·3160	2·1160	**1·9680**	3·2500
10	11·7240	2·4140	8·7160	3·8980	2·1600	2·0160	**1·8260**	3·0100
20	11·7120	1·6120	6·2140	2·5120	1·4400	1·4320	**1·3400**	2·0120
40	11·7120	1·2680	4·8180	1·8820	1·1260	1·2440	**1·0340**	1·4480
80	11·6920	1·1340	4·3720	1·7020	1·0260	1·1240	**0·6300**	1·0360
160	11·7019	1·0960	4·1880	1·6460	0·9980	1·1320	**0·4320**	0·7860
320	11·6900	1·0440	4·1820	1·6420	1·0140	1·1500	**0·3740**	0·6860
640	11·6919	1·0980	4·1360	1·6380	1·0320	1·1500	**0·4400**	0·6240

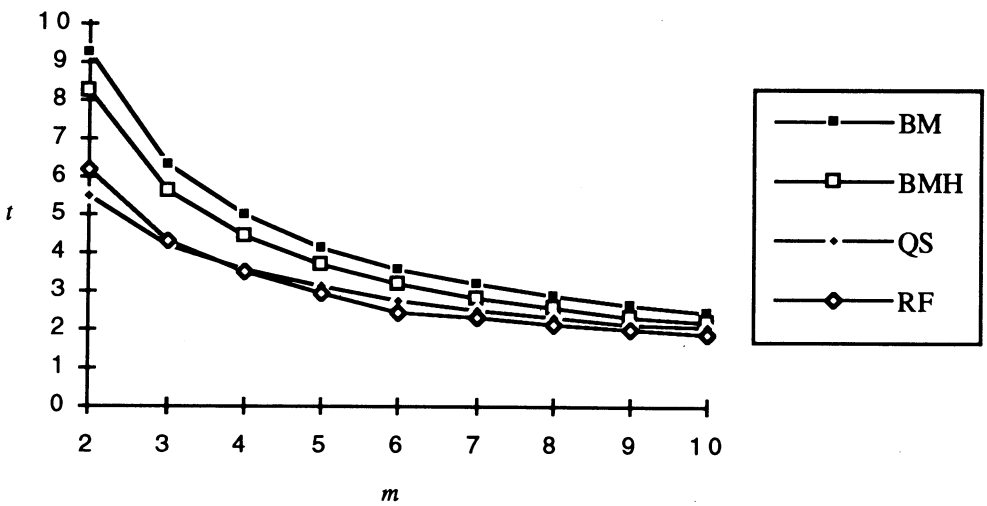

Figure 38. Running times for an alphabet of size 20—short patterns

22. W. Rytter, 'A correct preprocessing algorithm for Boyer–Moore string-searching', *SIAM J. Comput.* **9**, 509–512 (1980).
23. G. de V. Smit, 'A comparison of three string matching algorithms', *Software—Practice and Experience*, **12**, 57–66 (1982).
24. L.J. Guibas and A.M. Odlyzko, 'A new proof of the linearity of the Boyer–Moore string searching algorithm', *SIAM J. Comput.*, **9**, 672–682 (1980).
25. R. Cole, 'Tight bounds on the complexity of the Boyer–Moore string matching algorithm', *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990, pp. 224–233.
26. M. Crochemore, L. Gasieniec and W. Rytter, 'Turbo-BM', *Report LITP 92.61*, Université Paris 7, 1992.
27. A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M.T. Chen and J. Seiferas, 'The smallest automaton recognizing the subwords of a text', *Theoret. Comput. Sci.*, **40**, 31–55 (1985).
28. M. Crochemore, 'Transducers and repetitions', *Theoret. Comput. Sci.*, **45**, 63–86 (1986).

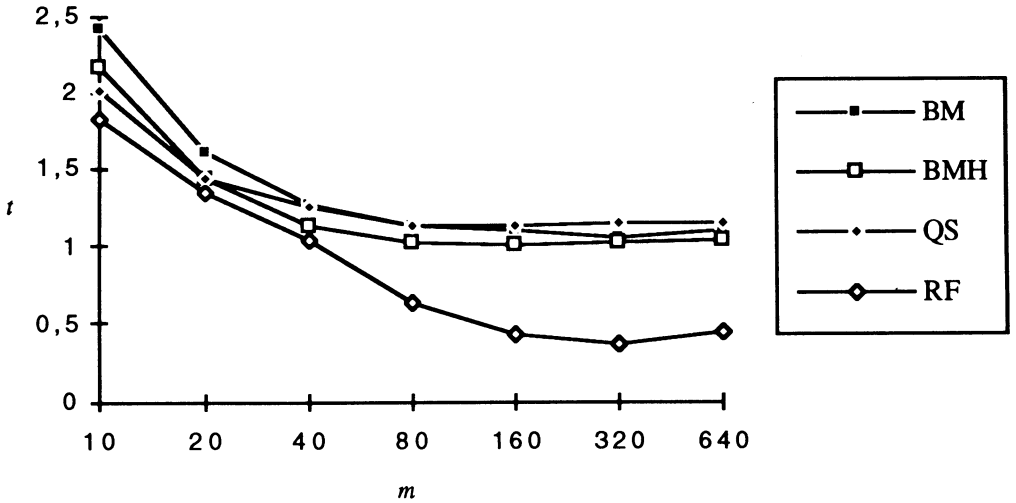

Figure 39. Running times for an alphabet of size 20—long patterns

Table XVI. Running times for an English text

m	BF	BM	AG	TBM	BMH	QS	RF	TRF
2	3·2100	2·8580	5·9960	4·6040	2·4840	**1·6340**	2·1920	3·4520
3	3·1420	1·9420	4·4560	3·2200	1·7680	**1·3260**	1·4560	2·3720
4	3·1960	1·5160	3·7080	2·4940	1·3560	**1·0800**	1·1700	1·8540
5	3·2040	1·2880	3·3540	2·0860	1·1320	**0·9420**	0·9640	1·5340
6	3·1680	1·1060	3·2140	1·8260	1·0160	**0·8280**	0·8540	1·3600
7	3·1720	0·9980	3·0300	1·5920	0·8860	**0·7640**	0·7920	1·1740
8	3·1920	0·8920	2·9620	1·4400	0·8180	0.7080	**0·6960**	1·0740
9	3·1940	0·8200	2·8480	1·3360	0·7500	0·6700	**0·6340**	1·0000
10	3·1940	0·7660	2·7540	1·2400	0·6980	0·6340	**0·5860**	0·9120
20	3·2000	0·4760	2·0360	0·7700	0·4580	0·4100	**0·3780**	0·5780
40	3·1800	0·3360	1·5460	0·5320	0·3260	0·3020	**0·2860**	0·3600
80	3·1760	0·2500	1·1620	0·3820	0·2360	0·2320	**0·1940**	0·2400
160	3·1780	0·1900	0·8940	0·2960	**0·1860**	0·1900	0·1940	0·2520
320	3·1820	0·1760	0·7520	0·2620	**0·1540**	0·1620	0·2560	0·3320
640	3·1640	0·1620	0·6780	0·2440	**0·1360**	0·1440	0·4220	0·6160

29. A.C. Yao, ‘The complexity of pattern matching for a random string’, *SIAM J. Comput.*, **8**, 368–387 (1979).

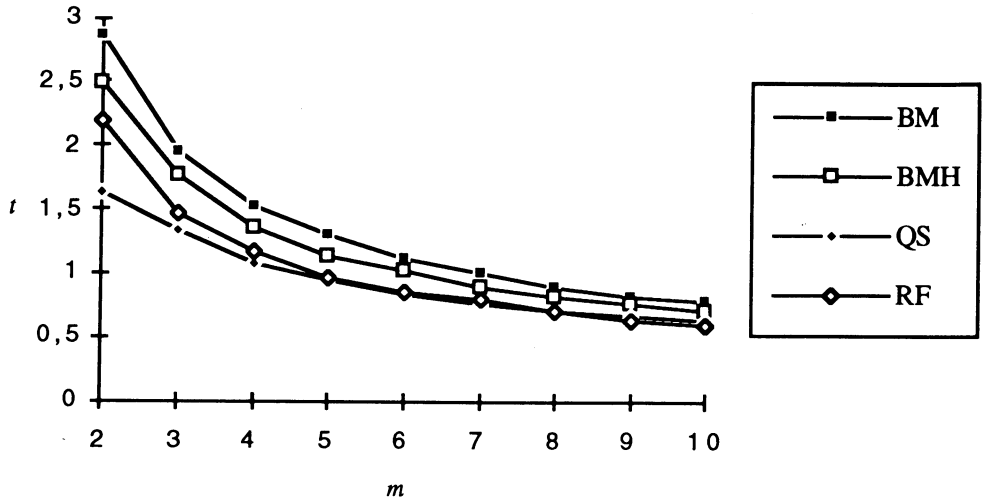

Figure 40. Running times for an English text—short patterns

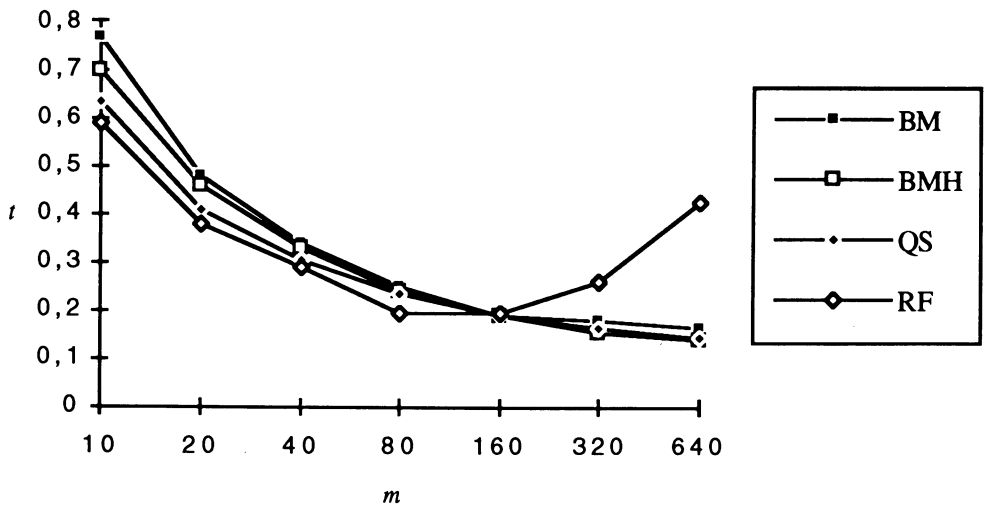

Figure 41. Running times for an English text—long patterns

Table XVII. Running times for C code

m	BF	BM	AG	TBM	BMH	QS	RF	TRF
2	3·0580	2·6240	5·6060	4·2620	2·3440	**1·5800**	2·1240	3·1560
3	3·0959	1·8060	4·1680	2·9340	1·6380	**1·3080**	1·4560	2·1120
4	3·0540	1·4580	3·5060	2·3700	1·3080	**1·0920**	1·1500	1·7800
5	3·0240	1·2660	3·2580	1·9740	1·2080	1·0120	**0·9840**	1·4300
6	3·0540	1·1000	4·0540	1·6760	1·0740	0·8780	**0·8660**	1·2800
7	3·0700	1·0240	2·9580	1·5060	0·9720	0·8260	**0·7560**	1·1260
8	3·1040	0·9780	2·8760	1·3440	0·8740	0·7720	**0·6860**	1·0260
9	3·0820	0·8440	2·6560	1·2460	0·8700	0·7420	**0·6400**	0·9340
10	3·1560	0·8180	2·5340	1·1680	0·8160	0·6680	**0·5820**	0·8640
20	3·2000	0·6240	1·9120	0·7120	0·6660	0·5740	**0·3640**	0·5680
40	3·3720	0·5060	1·3520	0·4420	0·6240	0·4720	**0·2440**	0·3560
80	3·4560	0·1880	0·9100	0·2860	0·4380	0·1980	**0·1740**	0·2640
160	3·4540	0·1260	0·5940	0·1780	0·3400	0·1480	**0·1460**	0·2760
320	3·4380	0·1060	0·6460	0·1360	0·2160	**0·1100**	0·2060	0·3300
640	3·4400	0·1020	0·3820	0·1360	0·2100	**0·0900**	0·3540	0·6060

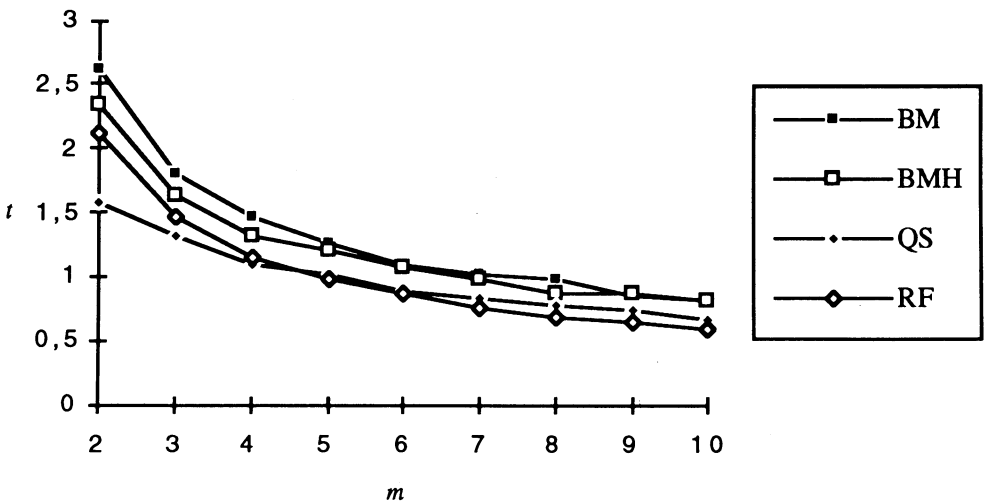

Figure 42. Running times for C code—short patterns

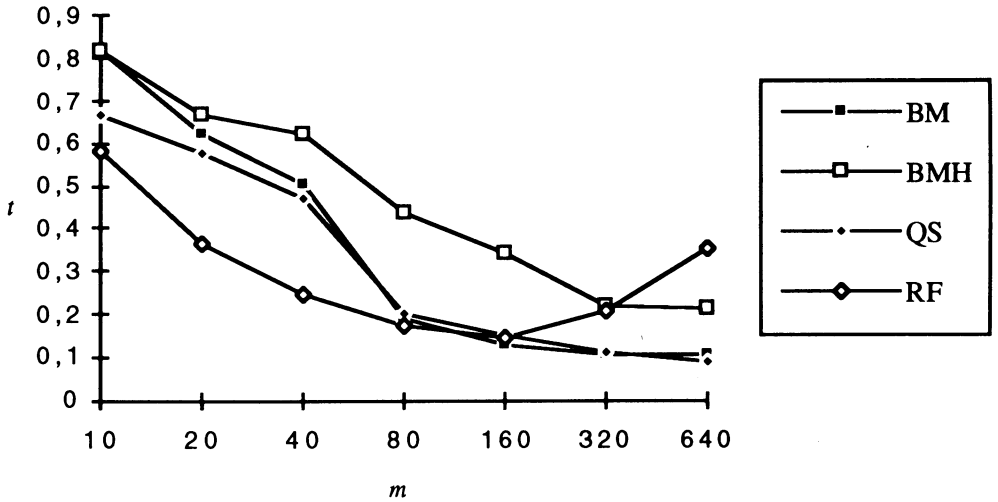

Figure 43. Running times for C code—long patterns

Table XVIII. Running times for a genome

m	BF	BM	AG	TBM	BMH	QS	RF	TRF
2	4·3820	4·1459	8·3660	6·3020	3·7320	**2·7700**	4·0220	5·7840
3	4·4540	3·0420	6·5200	4·8620	2·8460	**2·3500**	2·9500	4·3980
4	4·3540	2·5320	5·8340	4·0800	2·4180	**2·1920**	2·3140	3·5420
5	4·3880	2·2180	5·4180	3·5640	2·2140	**1·9600**	1·9780	2·9620
6	4·3600	1·9700	5·0340	3·2140	2·0400	1·8180	**1·6540**	2·5520
7	4·3920	1·8560	4·8200	2·9960	1·9260	1·7340	**1·4760**	2·2800
8	4·3840	1·7440	4·7240	2·7940	1·8680	1·6740	**1·3480**	2·0680
9	4·3740	1·6880	4·5780	2·7000	1·8200	1·6500	**1·2440**	1·9260
10	4·3520	1·6440	4·4880	2·6360	1·7940	1·6080	**1·1360**	1·8000
20	4·3660	1·4020	3·9560	2·2220	1·7440	1·6900	**0·7620**	1·1280
40	4·3560	1·2280	3·5620	1·9560	1·7300	1·6120	**0·4920**	0·7260
80	4·3520	1·0780	3·1760	1·7180	1·7460	1·5860	**0·3400**	0·4740
160	4·3460	0·9360	2·6900	1·4860	1·7040	1·6100	**0·3040**	0·4420
320	4·3600	0·8420	2·4740	1·3580	1·8200	1·6240	**0·2640**	0·5120
640	4·3560	0·8220	2·3480	1·2940	1·7460	1·6000	**0·5140**	0·7180

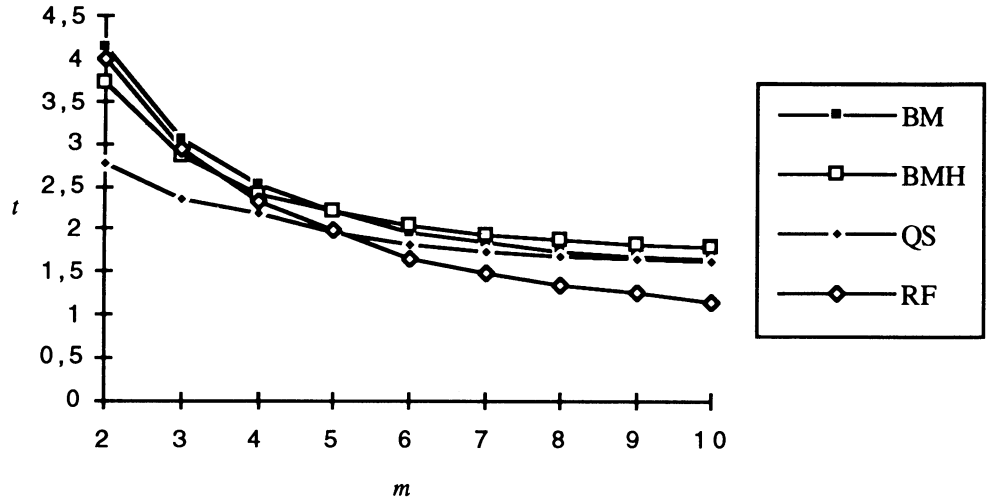

Figure 44. Running times for a genome–short patterns

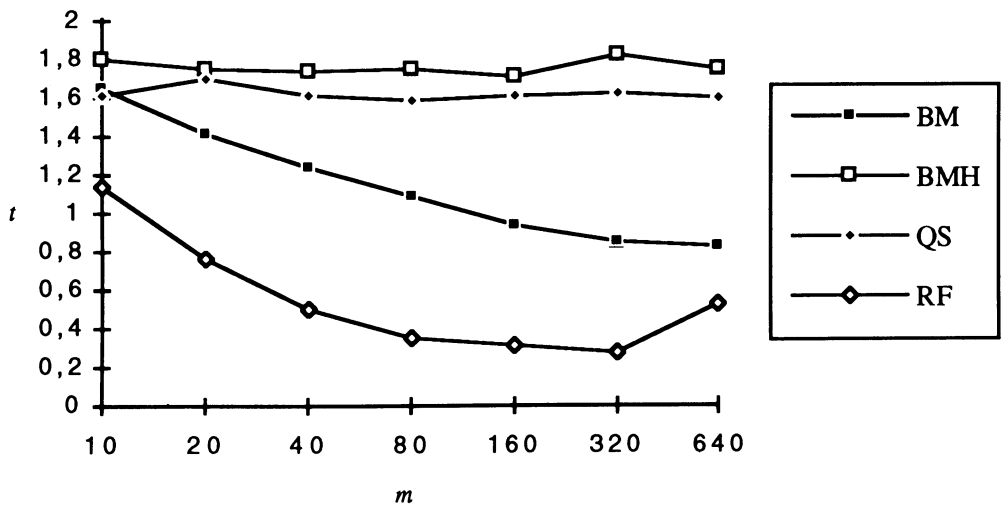

Figure 45. Running times for a genome—long patterns